Paxos Consensus, Deconstructed and Abstracted
Extended Version

Álvaro García-Pérez[1], Alexey Gotsman[1], Yuri Meshman[1], and Ilya Sergey[2]

[1] IMDEA Software Institute, Spain
{alvaro.garcia.perez,alexey.gotsman,yuri.meshman}@imdea.org
[2] University College London, UK
i.sergey@ucl.ac.uk

Abstract Lamport's Paxos algorithm is a classic consensus protocol for state machine replication in environments that admit crash failures. Many versions of Paxos exploit the protocol's intrinsic properties for the sake of gaining better run-time performance, thus widening the gap between the original description of the algorithm, which was proven correct, and its real-world implementations. In this work, we address the challenge of specifying and verifying complex Paxos-based systems by (a) devising composable specifications for implementations of Paxos's single-decree version, and (b) engineering disciplines to reason about protocol-aware, semantics-preserving optimisations to single-decree Paxos. In a nutshell, our approach elaborates on the deconstruction of single-decree Paxos by Boichat et al. We provide novel non-deterministic specifications for each module in the deconstruction and prove that the implementations refine the corresponding specifications, such that the proofs of the modules that remain unchanged can be reused across different implementations. We further reuse this result and show how to obtain a verified implementation of Multi-Paxos from a verified implementation of single-decree Paxos, by a series of novel protocol-aware transformations of the network semantics, which we prove to be behaviour-preserving.

1 Introduction

Consensus algorithms are an essential component of the modern fault-tolerant deterministic services implemented as message-passing distributed systems. In such systems, each of the distributed nodes contains a replica of the system's state (*e.g.*, a database to be accessed by the system's clients), and certain nodes may propose values for the next state of the system (*e.g.*, requesting an update in the database). Since any node can crash at any moment, all the replicas have to keep copies of the state that are consistent with each other. To achieve this, at each update to the system, all the non-crashed nodes run an instance of a *consensus protocol*, uniformly deciding on its outcome. The safety requirements for consensus can be thus stated as follows: "only a single value is decided uniformly by all non-crashed nodes, it never changes in the future, and the decided value has been proposed by some node participating in the protocol" [16].

The Paxos algorithm [15, 16] is the classic consensus protocol, and its single-decree version (SD-Paxos for short) allows a set of distributed nodes to reach an agreement on the outcome of a *single* update. Optimisations and modifications to SD-Paxos are common. For instance, the multi-decree version, often called Multi-Paxos [15, 27], considers multiple slots (*i.e.*, multiple positioned updates) and decides upon a result for *each* slot, by running a slot-specific instance of an SD-Paxos. Even though it is customary to think of Multi-Paxos as of a series of independent SD-Paxos instances, in reality the implementation features multiple protocol-aware optimisations, exploiting intrinsic dependencies between separate single-decree consensus instances to achieve better throughput. To a great extent, these and other optimisations to the algorithm are pervasive, and verifying a modified version usually requires to devise a new protocol definition and a proof from scratch. New versions are constantly springing (*cf.* Section 5 of [27] for a comprehensive survey) widening the gap between the description of the algorithms and their real-world implementations.

We tackle the challenge of *specifying* and *verifying* these distributed algorithms by contributing two verification techniques for consensus protocols.

Our first contribution is a family of composable specifications for Paxos' core subroutines. Our starting point is the deconstruction of SD-Paxos by Boichat *et al.* [2, 3], allowing one to consider a distributed consensus instance as a *shared-memory concurrent program*. We introduce novel specifications for Boichat *et al.*'s modules, and let them be non-deterministic. This might seem as an unorthodox design choice, as it *weakens* the specification. To show that our specifications are still *strong enough*, we restore the top-level *deterministic* abstract specification of the consensus, which is convenient for client-side reasoning. The weakness introduced by the non-determinism in the specifications has been impelled by the need to prove that the implementations of Paxos' components *refine* the specifications we have ascribed [9]. We prove the refinements modularly via the Rely/Guarantee reasoning with prophecy variables and explicit linearisation points [11, 26]. On the other hand, this weakness becomes a virtue when better understanding the volatile nature of Boichat *et al.*'s abstractions and of the Paxos algorithm, which may lead to newer modifications and optimisations.

Our second contribution is a methodology for verifying composite consensus protocols by reusing the proofs of their constituents, targeting specifically Multi-Paxos. We do so by distilling protocol-aware system optimisations into a separate semantic layer and showing how to obtain the realistic Multi-Paxos implementation from SD-Paxos by a *series of transformations* to the *network semantics* of the system, as long as these transformations preserve the behaviour observed by clients. We then provide a family of such transformations along with the formal conditions allowing one to compose them in a behaviour-preserving way.

We validate our approach for construction of modularly verified consensus protocols by providing an executable proof-of-concept implementation of Multi-Paxos with a high-level shared memory-like interface, obtained via a series of behaviour-preserving network transformations. The full proofs of lemmas and

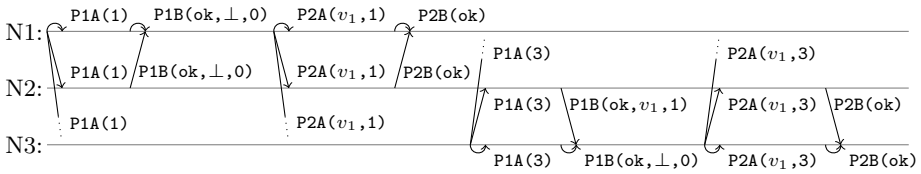


Figure 1. A run of SD-Paxos.

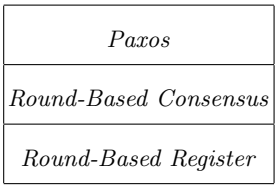
theorems from our development, as well as some boilerplate definitions, are given in the appendices at the end of the paper.

2 The Single-Decree Paxos Algorithm

We start with explaining SD-Paxos through an intuitive scenario. In SD-Paxos, each node in the system can adopt the roles of *proposer* or *acceptor*, or both. A value is decided when a *quorum* (*i.e.*, a majority of acceptors) accepts the value proposed by some proposer. Now consider a system with three nodes N1, N2 and N3, where N1 and N3 are both proposers and acceptors, and N2 is an acceptor, and assume N1 and N3 propose values v_1 and v_3, respectively.

The algorithm works in two phases. In Phase 1, a proposer polls every acceptor in the system and tries to convince a quorum to promise that they will later accept its value. If the proposer succeeds in Phase 1 then it moves to Phase 2, where it requests the acceptors to fulfil their promises in order to get its value decided. In our example, it would seem in principle possible that N1 and N3 could respectively convince two different quorums—one consisting of N1 and N2, and the other consisting of N2 and N3—to go through both phases and to respectively accept their values. This would happen if the communication between N1 and N3 gets lost and if N2 successively grants the promise and accepts the value of N1, and then does the same with N3. This scenario breaks the safety requirements for consensus because both v_1 and v_3—which can be different—would get decided. However, this cannot happen. Let us explain why.

The way SD-Paxos enforces the safety requirements is by distinguishing each attempt to decide a value with a unique *round*, where the rounds are totally ordered. Each acceptor stores its current round, initially the least one, and only grants a promise to proposers with a round greater or equal than its current round, at which moment the acceptor switches to the proposer's round. Figure 1 depicts a possible run of the algorithm. Assume that rounds are natural numbers, that the acceptors' current rounds are initially 0, and that the nodes N1 and N3 attempt to decide their values with rounds 1 and 3 respectively. In Phase 1, N1 tries to convince a quorum to switch their current round to 1 (messages `P1A(1)`). The message to N3 gets lost and the quorum consisting of N1 and N2 switches round and promises to only accept values at a round greater or equal than 1. Each acceptor that switches to the proposer's round sends back to the proposer its stored value and the round at which this value was accepted, or an undefined value if the acceptor never accepted any value yet (messages `P1B(ok,` $\perp$`,0)`, where $\perp$ denotes a default undefined value). After Phase 1, N1




```
1  val vP := undef;
2  proposeP(val v0) {
3    ⟨ assume(!(v0 = undef));
4      if (vP = undef) {
5        vP := v0;
6      } return vP; ⟩ }
```

Figure 2. Deconstruction of SD-Paxos (left) and specification of module *Paxos* (right).

picks as a candidate value the one accepted at the greatest round from those returned by the acceptors in the quorum, or its proposed value if all acceptors returned an undefined value. In our case, N1 picks its value v_1. In Phase 2, N1 requests the acceptors to accept the candidate value v_1 at round 1 (messages `P2A(`v_1`, 1)`). The message to N3 gets lost, and N1 and N2 accept value v_1, which gets decided (messages `P2B(ok)`).

Now N3 goes through Phase 1 with round 3 (messages `P1A(3)`). Both N2 and N3 switch to round 3. N2 answers N3 with its stored value v_1 and with the round 1 at which v_1 was accepted (message `P1B(ok,` v_1`, 1)`), and N3 answers itself with an undefined value, as it has never accepted any value yet (message `P1B(ok,` $\perp$`, 0)`). This way, if some value has been already decided upon, *any* proposer that convinces a quorum to switch to its round would receive the decided value from some of the acceptors in the quorum (recall that two quorums have a non-empty intersection). That is, N3 picks the v_1 returned by N2 as the candidate value, and in Phase 2 it manages that the quorum N2 and N3 accepts v_1 at round 3 (messages `P2A(`v_1`, 3)` and `P2B(ok)`). N3 succeeds in making a new decision, but the decided value remains the same, and, therefore, the safety requirements of a consensus protocol are satisfied.

3 The Faithful Deconstruction of SD-Paxos

We now recall the faithfull deconstruction of SD-Paxos in [2, 3], which we take as the reference architecture for the implementations that we aim to verify. We later show how each module of the deconstruction can be verified separately.

The deconstruction is depicted on the left of Figure 2, which consists of modules *Paxos*, *Round-Based Consensus* and *Round-Based Register*. These modules correspond to the ones in Figure 4 of [2], with the exception of *Weak Leader Election*. We assume that a correct process that is trusted by every other correct process always exists, and omit the details of the leader election. Leaders take the role of proposers and invoke the interface of *Paxos*. Each module uses the interface provided by the module below it.

The entry module *Paxos* implements SD-Paxos. Its specification (right of Figure 2) keeps a variable `vP` that stores the decided value (initially undefined) and provides the operation `proposeP` that takes a proposed value `v0` and returns `vP` if some value was already decided, or otherwise it returns `v0`. The code of the operation runs *atomically*, which we emphasise via angle brackets $\langle \ldots \rangle$. We define

this specification so it meets the safety requirements of a consensus, therefore, any implementation whose entry point refines this specification will have to meet the same safety requirements.

In this work we present both specifications and implementations in pseudo-code for an imperative WHILE-like language with basic arithmetic and primitive types, where `val` is some user-defined type for the values decided by Paxos, and `undef` is a literal that denotes an undefined value. The pseudo-code is self-explanatory and we restraint ourselves from giving formal semantics to it, which could be done in standard fashion if so wished [30]. At any rate, the pseudo-code is ultimately a vehicle for illustration and we stick to this informal presentation.

The implementation of the modules is depicted in Figures 3–5. We describe the modules following a bottom-up approach, which better fits the purpose of conveying the connection between the deconstruction and SD-Paxos. We start with module *Round-Based Register*, which offers operations `read` and `write` (Figure 3) and implements the replicated processes that adopt the role of acceptors (Figure 4). We adapt the wait-free, crash-stop implementation of *Round-Based Register* in Figure 5 of [2] by adding loops for the explicit reception of each individual message and by counting acknowledgement messages one by one. Processes are identified by integers from 1 to n, where n is the number of processes in the system. Proposers and acceptors exchange read and write requests, and their corresponding acknowledgements and non-acknowledgements. We assume a type `msg` for messages and let the message vocabulary to be as follows. Read requests `[RE, k]` carry the proposer's round `k`. Write requests `[WR, k, v]` carry the proposer's round `k` and the proposed value `v`. Read acknowledgements `[ackRE, k, v, k']` carry the proposer's round `k`, the acceptor's value `v`, and the round `k'` at which `v` was accepted. Read non-acknowledgements `[nackRE, k]` carry the proposer's round `k`, and so do carry write acknowledgements `[ackWR, k]` and write non-acknowledgements `[nackWR, K]`.

In the pseudo-code, we use `_` for a wildcard that could take any literal value. In the pattern-matching primitives, the literals specify the pattern against which an expression is being matched, and operator `@` turns a variable into a literal with the variable's value. Compare the `case [ackRE, @k, v, kW]:` in Figure 3, where the value of `k` specifies the pattern and `v` and `kW` get some values assigned, with the `case [RE, k]:` in Figure 4, where `k` gets some value assigned.

We assume the network ensures that messages are neither created, modified, deleted, nor duplicated, and that they are always delivered but with an arbitrarily large transmission delay.[3] Primitive `send` takes the destination `j` and the message `m`, and its effect is to send `m` from the current process to the process `j`. Primitive `receive` takes no arguments, and its effect is to receive at the current process a message `m` from origin `i`, after which it delivers the pair `(i, m)` of identifier and message. We assume that `send` is non-blocking and that `receive` blocks and suspends the process until a message is available, in which case the process awakens and resumes execution.

[3] We allow creation and duplication of `[RE, k]` messages in Section 5, where we obtain Multi-Paxos from SD-Paxos by a series of transformations of the network semantics.

```
1  read(int k) {
2    int j; val v; int kW; val maxV;
3    int maxKW; set of int Q; msg m;
4    for (j := 1, j <= n, j++)
5    { send(j, [RE, k]); }
6    maxKW := 0; maxV := undef; Q := {};
7    do { (j, m) := receive();
8         switch (m) {
9           case [ackRE, @k, v, kW]:
10            Q := Q ∪ {j};
11            if (kW >= maxKW)
12            { maxKW := kW; maxV := v; }
13          case [nackRE, @k]:
14            return (false, _);
15        } if (|Q| = ⌈(n+1)/2⌉)
16          { return (true, maxV); } }
17   while (true); }
18 write(int k, val vW) {
19   int j; set of int Q; msg m;
20   for (j := 1, j <= n, j++)
21   { send(j, [WR, k, vW]); }
22   Q := {};
23   do { (j, m) := receive();
24        switch (m) {
25          case [ackWR, @k]:
26            Q := Q ∪ {j};
27          case [nackWR, @k]:
28            return false;
29        } if (|Q| = ⌈(n+1)/2⌉)
30          { return true; } }
31   while (true); }
```

Figure 3. Implementation of *Round-Based Register* (`read` and `write`).

Each acceptor (Figure 4) keeps a value `v`, a current round `r` (called the *read round*), and the round `w` at which the acceptor's value was last accepted (called the *write round*). Initially, `v` is `undef` and both `r` and `w` are 0.

Phase 1 of SD-Paxos is implemented by operation `read` on the left of Figure 3. When a proposer issues a `read`, the operation requests each acceptor's promise to only accept values at a round greater or equal than `k` by sending `[RE, k]` (lines 4–5). When an acceptor receives a `[RE, k]` (lines 5–7 of Figure 4) it acknowledges the promise depending on its read round. If `k` is strictly less than `r` then the acceptor has already made a promise to another proposer with greater round and it sends `[nackRE, k]` back (line 8). Otherwise, the acceptor updates `r` to `k` and acknowledges by sending `[ackRE, k, v, w]` (line 9). When the proposer receives an acknowledgement (lines 8–10 of Figure 3) it counts acknowledgements up (line 10) and calculates the greatest write round at which the acceptors acknowledging so far accepted a value, and stores this value in `maxV` (lines 11–12). If a majority of acceptors acknowledged, the operation succeeds and returns `(true, maxV)` (lines 15–16). Otherwise, if the proposer received some `[nackRE, k]` the operation fails, returning `(false, _)` (lines 13–14).

Phase 2 of SD-Paxos is implemented by operation `write` on the right of Figure 3. After having collected promises from a majority of acceptors, the proposer picks the candidate value `vW` and issues a `write`. The operation requests each acceptor to accept the candidate value by sending `[WR, k, vW]` (lines 20–21). When an acceptor receives `[WR, k, vW]` (line 10 of Figure 4) it accepts the value depending on its read round. If `k` is strictly less than `r`, then the acceptor never promised to accept at such round and it sends `[nackWR, k]` back (line 11). Otherwise, the acceptor fullfils its promise and updates both `w` and `r` to `k` and assigns `vW` to its value `v`, and acknowledges by sending `[ackWR, k]` (line 12). Finally, when the proposer receives an acknowledgement (lines 23–25 of Fig-

```
1  process Acceptor(int j) {
2    val v := undef; int r := 0; int w := 0;
3    start() {
4      int i; msg m; int k;
5      do { (i, m) := receive();
6           switch (m) {
7             case [RE, k]:
8               if (k < r) { send(i, [nackRE, k]); }
9               else { ⟨ r := k; send(i, [ackRE, k, v, w]); ⟩ }
10            case [WR, k, vW]:
11              if (k < r) { send(i, [nackWR, k]); }
12              else { ⟨ r := k; w := k; v := vW; send(i, [ackWR, k]); ⟩ }
13          } }
14     while (true); } }
```

Figure 4. Implementation of *Round-Based Register* (acceptor).

```
1  proposeRC(int k, val v0) {
2    bool res; val v;
3    (res, v) := read(k);
4    if (res) {
5      if (v = undef) { v := v0; }
6      res := write(k, v);
7      if (res) { return (true, v); } }
8    return (false, _); }
```

```
1  proposeP(val v0) {
2    int k; bool res; val v;
3    k := pid();
4    do { (res, v) :=
5           proposeRC(k, v0);
6         k := k + n;
7    } while (!res);
8    return v; }
```

Figure 5. Implementation of *Round-Based Consensus* (left) and *Paxos* (right)

ure 3) it counts acknowledgements up (line 26) and checks whether a majority of acceptors acknowledged, in which case `vW` is decided and the operation succeeds and returns `true` (lines 29–30). Otherwise, if the proposer received some `[nackWR, k]` the operation fails and returns `false` (lines 27–28).[4]

Next, we describe module *Round-Based Consensus* on the left of Figure 5. The module offers an operation `proposeRC` that takes a round `k` and a proposed value `v0`, and returns a pair `(res, v)` of Boolean and value, where `res` informs of the success of the operation and `v` is the decided value in case `res` is `true`. We have taken the implementation from Figure 6 in [2] but adapted to our pseudo-code conventions. *Round-Based Consensus* carries out Phase 1 and Phase 2 of SD-Paxos as explained in Section 2. The operation `proposeRC` calls `read` (line 3) and if it succeeds then chooses a candidate value between the proposed value `v0` or the value `v` returned by `read` (line 5). Then, the operation calls `write` with the candidate value and returns `(true, v)` if `write` succeeds, or fails and returns `(false, _)` (line 8) if either the `read` or the `write` fails.

[4] For the implementation to be correct with our shared-memory-concurrency approach, the update of the data in acceptors must happen atomically with the sending of acknowledgements in lines 9 and 12 of Figure 4.

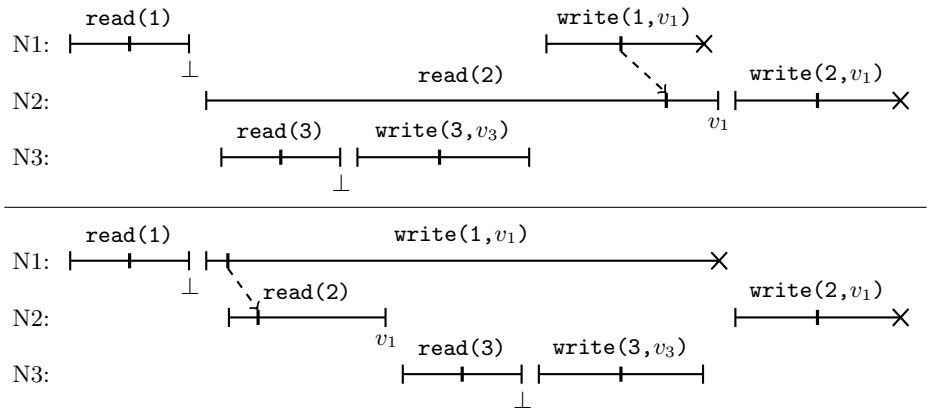


Figure 6. Two histories in which a failing `write` contaminates some acceptor.

Finally, the entry module *Paxos* on the right of Figure 5 offers an operation `proposeP` that takes a proposed value `v0` and returns the decided value. We assume that the system primitive `pid()` returns the process identifier of the current process. We have come up with this straightforward implementation of operation `proposeP`, which calls `proposeRC` with increasing round until the call succeeds, starting at a round equal to the process identifier `pid()` and increasing it by the number of processes n in each iteration. This guarantees that the round used in each invocation to `proposeRC` is unique.

The Challenge of Verifying the Deconstruction of Paxos. Verifying each module of the deconstruction separately is cumbersome because of the distributed character of the algorithm and the nature of a linearisation proof. A process may not be aware of the information that will flow from itself to other processes, but this future information flow may dictate whether some operation has to be linearised at the present. Figure 6 illustrates this challenge.

Let N1, N2 and N3 adopt both the roles of acceptors and proposers, which propose values v_1, v_2 and v_3 with rounds 1, 2 and 3 respectively. Consider the history on the top of the figure. N2 issues a read with round 2 and gets acknowledgements from all but one acceptors in a quorum. (Let us call this one acceptor A.) None of these acceptors have accepted anything yet and they all return $\perp$ as the last accepted value at round 0. In parallel, N3 issues a read with round 3 (third line in the figure) and gets acknowledgements from a quorum in which A does not occur. This read succeeds as well and returns `(true, undef)`. Then N3 issues a write with round 3 and value v_3. Again, it gets acknowledgements from a quorum in which A does not occur, and the write succeeds deciding value v_3 and returns `true`. Later on, and in real time order with the write by N3 but in parallel with the read by N2, node N1 issues a write with round 1 and value v_1 (first line in the figure). This write is to fail because the value v_3 was already decided with round 3. However, the write manages to "contaminate" acceptor A with value v_1, which now acknowledges N2 and sends v_1 as its last accepted value at round 1. Now N2 has gotten acknowledgements from a quorum, and since the

other acceptors in the quorum returned 0 as the round of their last accepted value, the read will catch value v_1 accepted at round 1, and the operation succeeds and returns `(true,` v_1`)`. This history linearises by moving N2's read after N1's write, and by respecting the real time order for the rest of the operations. (The linearisation ought to respect the information flow order between N1 and N2 as well, *i.e.*, N1 contaminates A with value v_1, which is read by N2.)

In the figure, a segment ending in an $\times$ indicates that the operation fails. The value returned by a successful read operation is depicted below the end of the segment. The linearisation points are depicted with a thick vertical line, and the dashed arrow indicates that two operations are in the information flow order.

The variation of this scenario on the bottom of Figure 6 is also possible, where N1's write and N2's read happen concurrently, but where N2's read is shifted backwards to happen before in real time order with N3's read and write. Since N1's write happens before N2's read in the information flow order, then N1's write has to inexorably linearise before N3's operations, which are the ones that will "steal" N1's valid round.

These examples give us three important hints for designing the specifications of the modules. First, after a decision is committed it is *not enough* to store only the decided value, since a posterior write may contaminate some acceptor with a value different from the decided one. Second, a read operation *may succeed* with some round even if by that time other operation has already succeeded with a higher round. And third, a write with a valid round *may fail* if its round will be "stolen" by a concurrent operation. The non-deterministic specifications that we introduce next allow one to model execution histories as the ones in Figure 6.

4 Modularly Verifying SD-Paxos

In this section, we provide non-deterministic specifications for *Round-Based Consensus* and *Round-Based Register* and show that each implementation refines its specification [9]. To do so, we instrument the implementations of all the modules with *linearisation-point* annotations and use Rely/Guarantee reasoning [26].

This time we follow a top-down order and start with the entry module *Paxos*.

Module *Paxos*. In order to prove that the implementation on the right of Figure 5 refines its specification on the right of Figure 2, we introduce the instrumented implementation in Figure 7, which uses the helping mechanism for external linearisation points of [18]. We assume that each proposer invokes `proposeP` with a unique proposed value. The auxiliary pending thread pool `ptp[`n`]` is an array of pairs of Booleans and values of length n, where n is the number of processes in the system. A cell `ptp[`i`]` containing a pair `(true,` v`)` signals that the process i proposed value v and the invocation `proposeP(`v`)` by process i awaits to be linearised. Once this invocation is linearised, the cell `ptp[`i`]` is updated to the pair `(false,` v`)`. A cell `ptp[`i`]` containing `undef` signals that the process i never proposed any value yet. The array `abs_resP[`n`]` of Boolean single-assignment variables stores the abstract result of each proposer's invocation. A linearisation-point annotation `lin(`i`)` takes a process identifier i and performs atomically the abstract operation invoked by proposer i and

```
1  (bool × val) ptp[1..n] := undef;
2  val abs_vP := undef; single bool abs_resP[1..n] := undef;
3  proposeP(val v0) {
4    int k; bool res; val v; assume(!(v0 = undef));
5    k := pid(); ptp[pid()] := (true, v0);
6    do { ⟨ (res, v) := proposeRC(k, v0);
7           if (res) {
8             for (i := 1, i <= n, i++) {
9               if (ptp[i] = (true, v)) { lin(i); ptp[i] := (false, v); } }
10            if (!(v = v0)) { lin(pid()); ptp[pid()] := (false, v0); } } ⟩
11       k := k + n; }
12   while (!res); return v; }
```

Figure 7. Instrumented implementation of *Paxos*.

assigns its result to `abs_resP[`i`]`. The abstract state is modelled by variable `abs_vP`, which corresponds to variable `vP` in the specification on the right of Figure 2. One invocation of `proposeP` may help linearise other invocations as follows. The linearisation point is together with the invocation to `proposeRC` (line 6). If `proposeRC` committed with some value `v`, the instrumented implementation traverses `ptp` and linearises all the proposers which were proposing value `v` (the proposer may linearise itself in this traversal) (lines 8–9). Then, the current proposer linearises itself if its proposed value `v0` is different from `v` (line 10), and the operation returns `v` (line 12). All the annotations and code in lines 6–10 are executed inside an atomic block, together with the invocation to `proposeRC(k, v0)`.

Theorem 1. *The implementation of Paxos on the right of Figure 5 linearises with respect to its specification on the right of Figure 2.*

Module *Round-Based Consensus*. The top of Figure 8 shows the non-deterministic module's specification. Global variable `vRC` is the decided value, initially `undef`. Global variable `roundRC` is the highest round at which some value was decided, initially 0; a global set of values `valsRC` (initially empty) contains values that may have been proposed by proposers. The specification is non-deterministic in that local value `vD` and Boolean `b` are unspecified, which we model by assigning random values to them. We assume that the current process identifier is $((\mathtt{k} - 1) \bmod n) + 1$, which is consistent with how rounds are assigned to each process and incremented in the code of `proposeP` on the right of Figure 5. If the unspecified value `vD` is neither in the set `valsRC` nor equal to `v0` then the operation returns `(false, _)` (line 11). This models that the operation fails without contaminating any acceptor. Otherwise, the operation may contaminate some acceptor and the value `vD` is added to the set `valsRC` (line 6). Now, if the unspecified Boolean `b` is false, then the operation returns `(false, _)` (lines 7 and 10), which models that the round will be stolen by a posterior operation. Finally, the operation succeeds if `k` is greater or equal than

```
1  val vRC := undef; int roundRC := 0; set of val valsRC := {};
2  proposeRC(int k, val v0) {
3    ⟨ val vD := random(); bool b := random();
4      assume(!(v0 = undef)); assume(pid() = ((k - 1) mod n) + 1);
5      if (vD ∈ (valsRC ∪ {v0})) {
6        valsRC := valsRC ∪ {vD};
7        if (b && (k >= roundRC)) { roundRC := k;
8                                   if (vRC = undef) { vRC := vD; }
9                                   return (true, vRC); }
10       else { return (false, _); } }
11     else { return (false, _); } ⟩ }
```

```
1  val abs_vRC := undef; int abs_roundRC := 0;
2  set of val abs_valsRC := {};
3  proposeRC(int k, val v0) {
4    single (bool × val) abs_resRC := undef; bool res; val v;
5    assume(!(v0 = undef)); assume(pid() = ((k - 1) mod n) + 1);
6    ⟨ (res, v) := read(k); if (res = false) { linRC(undef, _); } ⟩
7    if (res) { if (v = undef) { v := v0; }
8               ⟨ res := write(k, v); if (res) { linRC(v, true); }
9                                     else { linRC(v, false); } ⟩
10              if (res) { return (true, v); } }
11   return (false, _); }
```

Figure 8. Specification (top) and instrumented implementation (bottom) of *Round-Based Consensus*.

`roundRC` (line 7), and `roundRC` and `vRC` are updated and the operation returns `(true, vRC)` (lines 7–9).

In order to prove that the implementation in Figure 5 linearises with respect to the specification on the top of Figure 8, we use the instrumented implementation on the bottom of the same figure, where the abstract state is modelled by variables `abs_vRC`, `abs_roundRC` and `abs_valsRC` in lines 1–2, the local single-assignment variable `abs_resRC` stores the result of the abstract operation, and the linearisation-point annotations `linRC(vD, b)` take a value and a Boolean parameters and invoke the non-deterministic abstract operation and disambiguate it by assigning the parameters to the unspecified `vD` and `b` of the specification. There are two linearisation points together with the invocations of `read` (line 6) and `write` (line 8). If `read` fails, then we linearise forcing the unspecified `vD` to be `undef` (line 6), which ensures that the abstract operation fails without adding any value to `abs_valsRC` nor updating the round `abs_roundRC`. Otherwise, if `write` succeeds with value `v`, then we linearise forcing the unspecified value `vD` and Boolean `b` to be `v` and `true` respectively (line 8). This ensures that the abstract operation succeeds and updates the round `abs_roundRC` to `k` and assigns `v` to the decided value `abs_vRC`. If `write` fails then we linearise forcing the unspecified `vD` and `b` to be `v` and `false` respectively (line 9). This ensures that the abstract operation fails.

```
1  read(int k) {                                  16  val vRR := undef;
2    ⟨ val vD := random();                        17  int roundRR := 0;
3      bool b := random(); val v;                 18  set of val valsRR := {undef};
4      assume(vD ∈ valsRR);                       19
5      assume(pid() =                             20  write(int k, val vW) {
6        ((k - 1) mod n) + 1);                    21    ⟨ bool b := random();
7      if (b) {                                   22      assume(!(vW = undef));
8        if (k >= roundRR) {                      23      assume(pid() =
9          roundRR := k;                          24        ((k - 1) mod n) + 1);
10         if (!(vRR = undef)) {                  25      valsRR := valsRR ∪ {vW};
11           v := vRR; }                          26      if (b && (k >= roundRR)) {
12         else { v := vD; } }                    27        roundRR := k;
13       else { v := vD; }                        28        vRR := vW;
14       return (true, v); }                      29        return true; }
15     else { return (false, _); } ⟩ }           30      else { return false; } ⟩ }
```

Figure 9. Specification of *Round-Based Register*.

Theorem 2. *The implementation of Round-Based Consensus in Figure 5 linearises with respect to its specification on the top of Figure 8.*

Module *Round-Based Register*. Figure 9 shows the module's non-deterministic specification. Global variable `vRR` represents the decided value, initially `undef`. Global variable `roundRR` represents the current round, initially 0, and global set of values `valsRR`, initially containing `undef`, stores values that may have been proposed by some proposer. The specification is non-deterministic in that method `read` has unspecified local Boolean `b` and local value `vD` (we assume that `vD` is `valsRR`), and method `write` has unspecified local Boolean `b`. We assume the current process identifier is $((\texttt{k} - 1) \bmod n) + 1$.

Let us explain the specification of the `read` operation. The operation can succeed regardless of the proposer's round `k`, depending on the value of the unspecified Boolean `b`. If `b` is `true` and the proposer's round `k` is valid (line 8), then the read round is updated to `k` (line 9) and the operation returns `(true, v)` (line 14), where `v` is the read value, which coincides with the decided value if some decision was committed already or with `vD` otherwise. Now to the specification of operation `write`. The value `vW` is always added to the set `valsRR` (line 25). If the unspecified Boolean `b` is false (the round will be stolen by a posterior operation) or if the round `k` is non-valid, then the operation returns `false` (lines 26 and 30). Otherwise, the current round is updated to `k`, and the decided value `vRR` is updated to `vW` and the operation returns `true` (lines 27–29).

In order to prove that the implementation in Figures 3 and 4 linearises with respect to the specification in Figure 9, we use the instrumented implementation in Figures 10 and 11, which uses prophecy variables [1,26] that "guess" whether the execution of the method will reach a particular program location or not. The instrumented implementation also uses external linearisation points. In particular, the code of the acceptors may help to linearise some of the invocations to `read` and `write`, based on the prophecies and on auxiliary variables that count

the number of acknowledgements sent by acceptors after each invocation of a `read` or a `write`. The next paragraphs elaborate on our use of prophecy variables and on our helping mechanism.

Variables `abs_vRR`, `abs_roundRR` and `abs_valsRR` in Figure 10 model the abstract state. They are initially set to `undef`, 0 and the set containing `undef` respectively. Variable `abs_res_r[`k`]` is an infinite array of single-assignment pairs of Boolean and value that model the abstract results of the invocations to `read`. (Think of an infinite array as a map from integers to some type; we use the array notation for convenience.) Similarly, variable `abs_res_w[`k`]` is an infinite array of single-assignment Booleans that models the abstract results of the invocations to `write`. All the cells in both arrays are initially `undef` (*e.g.* the initial maps are empty). Variables `count_r[`k`]` and `count_w[`k`]` are infinite arrays of integers that model the number of acknowledgements sent (but not necessarily received yet) from acceptors in response to respectively read or write requests. All cells in both arrays are initially 0. The variable `proph_r[`k`]` is an infinite array of single-assignment pairs `bool` $\times$ `val`, modelling the prophecy for the invocations of `read`, and variable `proph_w[`k`]` is an infinite array of single-assignment Booleans modelling the prophecy for the invocations of `write`.

The linearisation-point annotations `linRE(k, vD, b)` for `read` take the proposer's round `k`, a value `vD` and a Boolean `b`, and they invoke the abstract operation and disambiguate it by assigning the parameters to the unspecified `vD` and `b` of the specification on the left of Figure 9. At the beginning of a `read(k)` (lines 11–14 of Figure 10), the prophecy `proph_r[k]` is set to `(true,` v`)` if the invocation reaches `PL: RE_SUCC` in line 26. The v is defined to coincide with `maxV` at the time when that location is reached. That is, v is the value accepted at the greatest round by the acceptors acknowledging so far, or undefined if no acceptor ever accepted any value. If the operation reaches `PL: RE_FAIL` in line 24 instead, the prophecy is set to `(false, _)`. (If the method never returns, the prophecy is left `undef` since it will never linearise.) A successful `read(k)` linearises in the code of the acceptor in Figure 11, when the $\lceil (n+1)/2 \rceil$th acceptor sends `[ackRE, k, v, w]`, and only if the prophecy is `(true,` v`)` and the operation was not linearised before (lines 10–14). We force the unspecified `vD` and `b` to be v and `true` respectively, which ensures that the abstract operation succeeds and returns `(true,` v`)`. A failing `read(k)` linearises at the `return` in the code of `read` (lines 23–24 of Figure 10), after the reception of `[nackRE, k]` from one acceptor. We force the unspecified `vD` and `b` to be `undef` and `false` respectively, which ensures that the abstract operation fails.

The linearisation-point annotations `linWR(k, vW, b)` for `write` take the proposer's round `k` and value `vW`, and a Boolean `b`, and they invoke the abstract operation and disambiguate it by assigning the parameter to the unspecified `b` of the specification on the right of Figure 9. At the beginning of a `write(k, vW)` (lines 31–33 of Figure 10), the prophecy `proph_r[k]` is set to `true` if the invocation reaches `PL: WR_SUCC` in line 45, or to `false` if it reaches `PL: WR_FAIL` in line 43 (or it is left `undef` if the method never returns). A successfully `write(k, vW)` linearises in the code of the acceptor in Figure 11, when the

```
1  val abs_vRR := undef; int abs_roundRR := 0;
2  set of val abs_valsRR := {undef};
3  single val abs_res_r[1..∞] := undef;
4  single val abs_res_w[1..∞] := undef;
5  int count_r[1..∞] := 0; int count_w[1..∞] := 0;
6  single (bool × val) proph_r[1..∞] := undef;
7  single bool proph_w[i..∞] := undef;
8  read(int k) {
9    int j; val v; set of int Q; int maxKW; val maxV; msg m;
10   assume(pid() = ((k - 1) mod n) + 1);
11   ⟨ if (operation reaches PL: RE_SUCC and define v = maxV at that time) {
12       proph_r[k] := (true, v); }
13     else { if (operation reaches PL: RE_FAIL) {
14              proph_r[k] := (false, _); } } ⟩
15   for (j := 1, j <= n, j++) { send(j, [RE, k]); }
16   maxKW := 0; maxV := undef; Q := {};
17   do { (j, m) := receive();
18        switch (m) {
19          case [ackRE, @k, v, kW]:
20            Q := Q ∪ {j};
21            if (kW >= maxKW) { maxKW := kW; maxV := v; }
22          case [nackRE, @k]:
23            ⟨ linRE(k, undef, false); proph_r[k] := undef;
24              return (false, _); ⟩ // PL: RE_FAIL
25        } if (|Q| = ⌈(n+1)/2⌉) {
26             return (true, maxV); } } // PL: RE_SUCC
27   while (true); }
28 write(int k, val vW) {
29   int j; set of int Q; msg m;
30   assume(!(vW = undef)); assume(pid() = ((k - 1) mod n) + 1);
31   ⟨ if (operation reaches PL: WR_SUCC) { proph_w[k] := true; }
32     else { if (operation reaches PL: WR_FAIL) {
33              proph_w[k] := false; } } ⟩
34   for (j := 1, j <= n, j++) { send(j, [WR, k, vW]); }
35   Q := {};
36   do { (j, m) := receive();
37        switch (m) {
38          case [ackWR, @k]:
39            Q := Q ∪ {j};
40          case [nackWR, @k]:
41            ⟨ if (count_w[k] = 0) {
42                linWR(k, vW, false); proph_w[k] := undef; }
43              return false; ⟩ // PL: WR_FAIL
44        } if (|Q| = ⌈(n+1)/2⌉) {
45             return true; } } // PL: WR_SUCC
46   while (true); }
```

Figure 10. Instrumented implementation of `read` and `write` methods.

```
1  process Acceptor(int j) {
2    val v := undef; int r := 0; int w := 0;
3    start() {
4      int i; msg m; int k;
5      do { (i, m) := receive();
6           switch (m) {
7             case [RE, k]:
8               if (k < r) { send(i, [nackRE, k]); }
9               else { ⟨ r := k;
10                       if (abs_res_r[k] = undef) {
11                         if (proph_r[k] = (true, v)) {
12                           if (count_r[k] = ⌈(n+1)/2⌉ - 1) {
13                             linRE(k, v, true); } } }
14                       count_r[k]++; send(i, [ackRE, k, v, w]); ⟩ }
15            case [WR, k, vW]:
16              if (k < r) { send(j, i, [nackWR, k]); }
17              else { ⟨ r := k; w := k; v := vW;
18                       if (abs_res_w[k] = undef) {
19                         if (!(proph_w[k] = undef)) {
20                           if (proph_w[k]) {
21                             if (count_w[k] = ⌈(n+1)/2⌉ - 1) {
22                               linWR(k, vW, true); } }
23                           else { linWR(k, vW, false); } } }
24                       count_w[k]++; send(j, i, [ackWR, k]); ⟩ }
25          } }
26     while (true); } }
```

Figure 11. Instrumented implementation of acceptor processes.

$\lceil (n+1)/2 \rceil$th acceptor sends `[ackWR, k]`, and only if the prophecy is `true` and the operation was not linearised before (lines 17–24). We force the unspecified `b` to be `true`, which ensures that the abstract operation succeeds deciding value `vW` and updates `roundRR` to `k`. A failing `write(k, vW)` may linearise either at the `return` in its own code (lines 41–43 of Figure 10) if the proposer received one `[nackWR, k]` and no acceptor sent any `[ackWR, k]` yet, or at the code of the acceptor, when the first acceptor sends `[ackWR, k]`, and only if the prophecy is `false` and the operation was not linearised before. In both cases, we force the unspecified `b` to be `false`, which ensures that the abstract operation fails.

Theorem 3. *The implementation of Round-Based Register in Figures 10 and 11 linearises with respect to its specification in Figure 9.*

5 Multi-Paxos via Network Transformations

We now turn to more complicated distributed protocols that build upon the idea of Paxos consensus. Our ultimate goal is to reuse the verification result from the Sections 3–4, as well as the high-level round-based register interface. In this section, we will demonstrate how to reason about an implementation

of Multi-Paxos as of an array of *independent* instances of the *Paxos* module defined previously, despite the subtle dependencies between its sub-components, as present in Multi-Paxos's "canonical" implementations [5, 15, 27]. While an abstraction of Multi-Paxos to an array of independent shared "single-shot" registers is almost folklore, what appears to be inherently difficult is to verify a Multi-Paxos-based consensus (*wrt.* to the array-based abstraction) by means of *reusing* the proof of a SD-Paxos. All proofs of Multi-Paxos we are aware of are, thus, *non-modular* with respect to underlying SD-Paxos instances [5,22,24], *i.e.*, they require one to redesign the invariants of the *entire* consensus protocol.

This proof modularity challenge stems from the optimised nature of a classical Multi-Paxos protocol, as well as its real-world implementations [6]. In this part of our work is to distil such protocol-aware optimisations into a separate *network semantics layer*, and show that each of them refines the semantics of a Cartesian product-based view, *i.e.*, exhibits the very same client-observable behaviours. To do so, we will establishing the refinement between the optimised implementations of Multi-Paxos and a simple Cartesian product abstraction, which will allow to extend the register-based abstraction, explored before in this paper, to what is considered to be a canonical amortised Multi-Paxos implementation.

5.1 Abstract Distributed Protocols

We start by presenting the formal definitions of encoding distributed protocols (including Paxos), their message vocabularies, protocol-based network semantics, and the notion of an observable behaviours.

Protocols and messages. Figure 12 provides basic definitions of the distributed protocols and their components. Each protocol p is a tuple $\langle \Delta, \mathcal{M}, \mathcal{S}_{\text{int}}, \mathcal{S}_{\text{rcv}}, \mathcal{S}_{\text{snd}} \rangle$. Δ is a set of local states, which can be assigned to each of the participating nodes, also determining the node's role via an additional tag,[5] if necessary (*e.g.*, an acceptor and a proposer states in Paxos are different). $\mathcal{M}$ is a "message vocabulary", determining the set of messages that can be used for communication between the nodes.

Protocols	$\mathcal{P} \ni p \triangleq \langle \Delta, \mathcal{M}, \mathcal{S} \rangle$
Configurations	$\Sigma \ni \sigma \triangleq \mathsf{Nodes} \rightharpoonup \Delta$
Internal steps	$\mathcal{S}_{\text{int}} \in \Delta \times \Delta$
Receive-steps	$\mathcal{S}_{\text{rcv}} \in \Delta \times \mathcal{M} \times \Delta$
Send-steps	$\mathcal{S}_{\text{snd}} \in \Delta \times \Delta \times \wp(\mathcal{M})$

Figure 12. States and transitions.

Messages can be thought of as JavaScript-like dictionaries, pairing unique fields (isomorphic to strings) with their values. For the sake of a uniform treatment, we assume that each message $m \in \mathcal{M}$ has at least two fields, *from* and *to* that point to the source and the destination node of a message, correspondingly. In addition to that, for simplicity we will assume that each message carries a Boolean field *active*, which is set to True when the message is sent and is set to False when the message is received by its destination node. This flag is required to keep history information about messages sent in the past, which is customary in frameworks for reasoning about distributed protocols [10, 23, 28]. We assume that a "message soup" M is a multiset of messages (*i.e.* a set with zero or more

[5] We leave out implicit the consistency laws for the state, that are protocol-specific.

$$\textsc{StepInt}\quad \frac{n \in \mathsf{dom}(\sigma) \qquad \delta = \sigma(n) \qquad \langle \delta, \delta' \rangle \in p.\mathcal{S}_{\text{int}} \qquad \sigma' = \sigma[n \mapsto \delta']}{\langle \sigma, M \rangle \xRightarrow[\text{int}]{p} \langle \sigma', M \rangle}$$

$$\textsc{StepSend}\quad \frac{n \in \mathsf{dom}(\sigma) \qquad \delta = \sigma(n) \qquad \langle \delta, \delta', \mathsf{ms} \rangle \in p.\mathcal{S}_{\text{snd}} \qquad \sigma' = \sigma[n \mapsto \delta'] \qquad M' = M \cup \mathsf{ms}}{\langle \sigma, M \rangle \xRightarrow[\text{snd}]{p} \langle \sigma', M' \rangle}$$

$$\textsc{StepReceive}\quad \frac{m \in M \qquad m.active \qquad m.to \in \mathsf{dom}(\sigma) \qquad \delta = \sigma(m.to) \qquad \langle \delta, m, \delta' \rangle \in p.\mathcal{S}_{\text{rcv}} \qquad m' = m[active \mapsto \mathsf{False}] \qquad \sigma' = \sigma[n \mapsto \delta'] \qquad M' = M \setminus \{m\} \cup \{m'\}}{\langle \sigma, M \rangle \xRightarrow[\text{rcv}]{p} \langle \sigma', M' \rangle}$$

Figure 13. Transition rules of the simple protocol-aware network semantics

copies of each message) and we consider that each copy of the same message in the multiset has its own "identity", and we write $m \neq m'$ to represent that m and m' are not the same copy of a particular message.

Finally, $\mathcal{S}_{\{\text{int,rcv,snd}\}}$ are step-relations that correspond to the internal changes in the local state of a node ($\mathcal{S}_{\text{int}}$), as well as changes associated with sending ($\mathcal{S}_{\text{snd}}$) and receiving ($\mathcal{S}_{\text{rcv}}$) messages by a node, as allowed by the protocol. Specifically, $\mathcal{S}_{\text{int}}$ relates a local node state before and after the allowed internal change; $\mathcal{S}_{\text{rcv}}$ relates the initial state and an incoming message $m \in \mathcal{M}$ with the resulting state; $\mathcal{S}_{\text{snd}}$ relates the internal state, the output state and the set of atomically sent messages. For simplicity we will assume that $\mathsf{id} \subseteq \mathcal{S}_{\text{int}}$.

In addition, we consider $\Delta_0 \subseteq \Delta$—the set of the allowed *initial* states, in which the system can be present at the very beginning of its execution. The global state of the network $\sigma \in \Sigma$ is a map from node identifiers ($n \in \mathsf{Nodes}$) to local states from the set of states Δ, defined by the protocol.

Simple network semantics. The simple initial operational semantics of the network ($\xRightarrow{p} \subseteq (\Sigma \times \wp(\mathcal{M})) \times (\Sigma \times \wp(\mathcal{M}))$) is parametrised by a protocol p and relates the initial *configuration* (*i.e.*, the global state and the set of messages) with the resulting configuration. It is defined via as a reflexive closure of the union of three relations $\xRightarrow[\text{int}]{p} \cup \xRightarrow[\text{rcv}]{p} \cup \xRightarrow[\text{snd}]{p}$, their rules are given in Figure 13.

The rule StepInt corresponds to a node n picked non-deterministically from the domain of a global state σ, executing an internal transition, thus changing its local state from δ to δ'. The rule StepReceive non-deterministically picks a m message from a message soup $M \subseteq \mathcal{M}$, changes the state using the protocol's receive-step relation $p.\mathcal{S}_{\text{rcv}}$ at the corresponding host node to, and updates its local state accordingly in the common mapping ($\sigma[to \mapsto \delta']$). Finally, the rule StepSend, non-deterministically picks a node n, executes a send-step, which results in updating its local state emission of a set of messages ms, which is added to the resulting soup. In order to "bootstrap" the execution, the initial states from the set $\Delta_0 \subseteq \Delta$ are assigned to the nodes.

We next define the observable protocol behaviours *wrt.* the simple network semantics as the prefix-closed set of all system's configuration traces.

Definition 1 (Protocol behaviours).

$$\mathcal{B}_p = \bigcup_{m\in\mathbb{N}} \left\{ \langle\langle \sigma_0, M_0\rangle, \ldots, \langle\sigma_m, M_m\rangle\rangle \;\middle|\; \begin{array}{r} \exists \delta_0^{n\in N} \in \Delta_0,\, \sigma_0 = \biguplus_{n\in N}[n \mapsto \delta_0^n] \wedge \\ \langle\sigma_0, M_0\rangle \overset{p}{\Longrightarrow} \ldots \overset{p}{\Longrightarrow} \langle\sigma_m, M_m\rangle \end{array} \right\}$$

That is, the set of behaviours captures all possible configurations of initial states for a fixed set of nodes $N \subseteq \mathsf{Nodes}$. In this case, the set of nodes N is an implicit parameter of the definition, which we fix in the remainder of this section.

Example 1 (Encoding SD-Paxos). An abstract distributed protocol for SD-Paxos can be extracted from the pseudo-code of Section 3 by providing a suitable small-step operational semantics à la Winskel [30]. We restraint ourselves from giving such formal semantics, but in Appendix D we outline how the distributed protocol would be obtained from the given operational semantics and from the code in Figures 3, 4 and 5.

5.2 Out-of-Thin-Air Semantics.

We now introduce an intermediate version of a simple protocol-aware semantics that generates messages "out of thin air" according to a certain predicate $\mathcal{P} \subseteq \Delta \times \mathcal{M}$, which determines whether the network generates a certain message without exercising the corresponding send-transition. The rule is as follows:

$$\frac{\begin{array}{l}\textsc{OTASend} \\ n \in \mathsf{dom}(\sigma) \qquad \delta = \sigma(n) \qquad \mathcal{P}(\delta, m) \qquad M' = M \cup \{m\}\end{array}}{\langle\sigma, M\rangle \xrightarrow[\text{ota}]{p,\mathcal{P}} \langle\sigma, M'\rangle}$$

That is, a random message m can be sent at any moment in the semantics described by $\overset{p}{\Longrightarrow} \cup \xrightarrow[\text{ota}]{p,\mathcal{P}}$, given that the node n, "on behalf of which" the message is sent is in a state δ, such that $\mathcal{P}(\delta, m)$ holds.

Example 2. In the context of Single-Decree Paxos, we can define $\mathcal{P}$ as follows:

$$\mathcal{P}(\delta, m) \triangleq m.\mathit{content} = \texttt{[RE,}\, k\texttt{]} \wedge \delta.\texttt{pid} = n \wedge \delta.\texttt{role} = \mathit{Proposer} \wedge k \leq \delta.\texttt{kP}$$

In other words, if a node n is a *Proposer* currently operating with a round $\delta.\texttt{kP}$, the network semantics can always send another request "on its behalf", thus generating the message "out-of-thin-air". Importantly, the last conjunct in the definition of $\mathcal{P}$ is in terms of $\leq$, rather than equality. This means that the predicate is intentionally loose, allowing for sending even "stale" messages, with expired rounds that are smaller than what n currently holds (no harm in that!).

By definition of single-decree Paxos protocol, the following lemma holds:

Lemma 1 (OTA refinement). $\mathcal{B}_{\overset{p}{\Longrightarrow} \cup \xrightarrow[ota]{p,\mathcal{P}}} \subseteq \mathcal{B}_p$, *where* p *is an instance of the module Paxos, as defined in Section 3 and in Example 1.*

$$\textsc{SRStepInt}\quad \frac{i \in I \qquad n \in \mathsf{dom}(\sigma) \qquad \delta = \sigma(n)[i] \qquad \langle \delta, \delta' \rangle \in p.\mathcal{S}_{\text{int}} \qquad \sigma' = \sigma[n[i] \mapsto \delta']}{\langle \sigma, M \rangle \xRightarrow[\text{int}]{\times} \langle \sigma', M \rangle}$$

$$\textsc{SRStepSend}\quad \frac{i \in I \qquad n \in \mathsf{dom}(\sigma) \qquad \delta = \sigma(n)[i] \qquad \langle \delta, \delta', \mathsf{ms} \rangle \in p.\mathcal{S}_{\text{snd}} \qquad \sigma' = \sigma[n[i] \mapsto \delta'] \qquad M' = M \cup \mathsf{ms}[\mathit{slot} \mapsto i]}{\langle \sigma, M \rangle \xRightarrow[\text{snd}]{\times} \langle \sigma', M' \rangle}$$

$$\textsc{SRStepReceive}\quad \frac{m \in M \quad m.\mathit{active} \qquad m.\mathit{to} \in \mathsf{dom}(\sigma) \qquad \delta = \sigma(m.\mathit{to})[m.\mathit{slot}] \qquad \langle \delta, m, \delta' \rangle \in p.\mathcal{S}_{\text{rcv}} \qquad m' = m[\mathit{active} \mapsto \mathsf{False}] \qquad \sigma' = \sigma(n)[m.\mathit{slot} \mapsto \delta'] \qquad M' = M \setminus \{m\} \cup \{m'\}}{\langle \sigma, M \rangle \xRightarrow[\text{rcv}]{\times} \langle \sigma', M' \rangle}$$

Figure 14. Transition rules of the slot-replicating network semantics.

5.3 Slot-Replicating Network Semantics.

With the basic definitions at hand, we now proceed to describing alternative network behaviours that make use of a specific protocol $p = \langle \Delta, \mathcal{M}, \mathcal{S}_{\text{int}}, \mathcal{S}_{\text{rcv}}, \mathcal{S}_{\text{snd}} \rangle$, which we will consider to be fixed for the remainder of this section, so we will be at times referring to its components (*e.g.*, $\mathcal{S}_{\text{int}}$, $\mathcal{S}_{\text{rcv}}$, *etc*) without a qualifier.

Figure 14 describes a semantics of a *slot-replicating* (SR) network that exercises multiple copies of the *same* protocol instance p_i for $i \in I$, some, possibly infinite, set of indices, to which we will be also referring as *slots*. Multiple copies of the protocol are incorporated by enhancing the messages from p's vocabulary $\mathcal{M}$ with the corresponding indices, and implementing the on-site dispatch of the indexed messages to corresponding protocol instances at each node. The local protocol state of each node is, thus, no longer a single element being updated, but rather an *array*, mapping $i \in I$ into δ_i—the corresponding local state component. The small-step relation for SR semantics is denoted by $\xRightarrow{\times}$. The rule SRStepInt is similar to StepInt of the simple semantics, with the difference that it picks not only a node but also an index i, thus referring to a specific component $\sigma(n)[i]$ as δ and updating it correspondingly ($\sigma(n)[i] \mapsto \delta'$). For the remaining transitions, we postulate that the messages from p's vocabulary $p.\mathcal{M}$ are enhanced to have a dedicated field *slot*, which indicates a protocol copy at a node, to which the message is directed. The receive-rule SRStepReceive is similar to StepReceive but takes into the account the value of $m.\mathit{slot}$ in the received message m, thus redirecting it to the corresponding protocol instance and updating the local state appropriately. Finally, the rule SRStepSend can be now executed for any slot $i \in I$, reusing most of the logic of the initial protocol and otherwise mimicking its simple network semantic counterpart StepSend.

Importantly, in this semantics, for two different slots i, j, such that $i \neq j$, the corresponding "projections" of the state behave *independently* from each other. Therefore, transitions and messages in the protocol instances indexed by i at different nodes *do not interfere* with those indexed by j. This observation can be stated formally. In order to do so we first defined the behaviours of slot-replicating networks and their projections as follows:

Definition 2 (Slot-replicating protocol behaviours).

$$\mathcal{B}_\times = \bigcup_{m\in\mathbb{N}} \left\{ \langle\langle\sigma_0, M_0\rangle, \ldots, \langle\sigma_m, M_m\rangle\rangle \;\middle|\; \begin{array}{l} \exists \delta_0^{n\in N} \in \Delta_0, \\ \sigma_0 = \biguplus_{n\in N}[n \mapsto \{i \mapsto \delta_0^n \mid i \in I\}] \wedge \\ \langle\sigma_0, M_0\rangle \xRightarrow{p} \ldots \xRightarrow{p} \langle\sigma_m, M_m\rangle \end{array} \right\}$$

That is, the slot-replicated behaviours are merely behaviours with respect to networks, whose nodes hold *multiple instances* of the same protocol, indexed by slots $i \in I$. For a slot $i \in I$, we define *projection* $\mathcal{B}_\times|_i$ as a set of global state traces, where each node's local states is restricted only to its ith component. The following simulation lemma holds naturally, connecting the state-replicating network semantics and simple network semantics.

Lemma 2 (Slot-replicating simulation). *For all $I, i \in I$, $\mathcal{B}_\times|_i = \mathcal{B}_p$.*

Example 3 (Slot-replicating semantics and Paxos). Given our representation of Paxos using roles (acceptors/proposers) encoded via the corresponding parts of the local state δ, we can construct a "naïve" version of Multi-Paxos by using the SR semantics for the protocol. In such, every slot will correspond to a SD Paxos instance, not interacting with any other slots. From the practical perspective, such an implementation is rather non-optimal, as it does not exploit dependencies between rounds accepted at different slots.

5.4 Widening Network Semantics.

We next consider a version of the SR semantics, extended with a new rule for handling received messages. In the new semantics, dubbed *widening*, a node, upon receiving a message $m \in T$, where $T \subseteq p.\mathcal{M}$, for a slot i, *replicates* it for all slots from the index set I, for the very same node. The new rule is as follows:

$$\textsc{WStepReceiveT}\quad \frac{\begin{array}{c} m \in M \qquad m.active \qquad m.to \in \mathsf{dom}(\sigma) \qquad \delta = \sigma(m.to)[m.slot] \\ \langle\delta, m, \delta'\rangle \in p.\mathcal{S}_{\mathrm{rcv}} \qquad m' = m[active \mapsto \mathsf{False}] \qquad \sigma' = \sigma(n)[m.slot \mapsto \delta'] \\ \mathsf{ms} = \text{if } (m \in T) \text{ then } \left\{m' \mid m' = m[slot \mapsto j], j \in I\right\} \text{ else } \emptyset \end{array}}{\langle\sigma, M\rangle \xRightarrow[\mathrm{rcv}]{\nabla} \langle\sigma', (M \setminus \{m\}) \cup \left\{m'\right\} \cup \mathsf{ms}\rangle}$$

At first, this semantics seems rather unreasonable: it might create more messages than the system can "consume". However, it is possible to prove that, under certain conditions on the protocol p, the set of behaviours observed under this semantics (*i.e.*, with SRStepReceive replaced by WStepReceiveT) is *not larger* than $\mathcal{B}_\times$ as given by Definition 2. To state this formally we first relate the set of "triggering" messages T from WStepReceiveT to a specific predicate $\mathcal{P}$.

Definition 3 (OTA-compliant message sets). *The set of messages $T \subseteq p.\mathcal{M}$ is OTA-compliant with the predicate $\mathcal{P}$ iff for any $b \in \mathcal{B}_p$ and $\langle\sigma, M\rangle \in b$, if $m \in M$, then $\mathcal{P}(\sigma(m.\mathtt{from}), m)$.*

In other words, the protocol p is relaxed enough to "justify" the presence of m in the soup at *any* execution, by providing the predicate $\mathcal{P}$, relating the message to the corresponding sender's state. Next, we use this definition to slot-replicating and widening semantics via the following definition.

Definition 4 ($\mathcal{P}$-monotone protocols). *A protocol p is $\mathcal{P}$-monotone iff for any, $b \in \mathcal{B}_\times$, $\langle \sigma, M\rangle \in b$, m, $i = m.slot$, and $j \neq i$, if $\mathcal{P}(\sigma(m.\texttt{from})[i], \natural m)$ then we have that $\mathcal{P}(\sigma(m.\texttt{from})[j], \natural m)$, where $\natural m$ "removes" the slot field from m.*

Less formally, Definition 4 ensures that in a slot-replicated product $\times$ of a protocol p, different components cannot perform "out of sync" *wrt.* $\mathcal{P}$. Specifically, if a node in ith projection is related to a certain message $\natural m$ via $\mathcal{P}$, then any other projection j of the same node will be $\mathcal{P}$-related to this message, as well.

Example 4. This is a "non-example". A version of slot-replicated SD-Paxos, where we allow for arbitrary increments of the round *per-slot* at a same proposer node (*i.e.*, out of sync), would not be monotone *wrt.* $\mathcal{P}$ from Example 2. In contrast, a slot-replicated product of SD-Paxos instances with fixed rounds is monotone *wrt.* the same $\mathcal{P}$.

Lemma 3. *If T from* WStepReceiveT *is OTA-compliant with predicate $\mathcal{P}$, such that $\mathcal{B}\xRightarrow{p}\cup\xRightarrow[ota]{p,\mathcal{P}} \subseteq \mathcal{B}\xRightarrow{p}$ and p is $\mathcal{P}$-monotone, then $\mathcal{B}\xRightarrow{\nabla} \subseteq \mathcal{B}\xRightarrow{\times}$.*

Example 5 (Widening semantics and Paxos). The SD-Paxos instance as described in Section 3 satisfies the refinement condition from Lemma 3. By taking $T = \{m \mid m = \{content = \texttt{[RE, k]}; \ldots\}\}$ and using Lemma 3, we obtain the refinement between widened semantics and SR semantics of Paxos.

5.5 Optimised Widening Semantics.

Our next step towards a realistic implementation of Multi-Paxos out of SD-Paxos instances is enabled by an observation that in the widening semantics, the replicated messages are *always* targeting the same node, to which the initial message $m \in T$ was addressed. This means that we can optimise the receive-step, making it possible to execute multiple receive-transitions of the core protocol in batch. The following rule OWStepReceiveT captures this intuition formally:

$$\text{OWStepReceiveT}\quad \frac{m \in M \qquad m.active \qquad m.to \in \mathsf{dom}(\sigma) \qquad \langle \sigma', \mathsf{ms}\rangle = \mathsf{receiveAndAct}(\sigma, n, m)}{\langle \sigma, M\rangle \xRightarrow[\text{rcv}]{\nabla^*} \langle \sigma', M \setminus \{m\} \cup \{m[active \mapsto \mathsf{False}]\} \cup \mathsf{ms}\rangle}$$

where $\mathsf{receiveAndAct}(\sigma, n, m) \triangleq \langle \sigma', \mathsf{ms}\rangle$, such that $\mathsf{ms} = \bigcup_j \{m[slot \mapsto j] \mid m \in \mathsf{ms}_j\}$, $\forall j \in I, \delta = \sigma(m.to)[j] \wedge \langle \delta_j, \natural m, \delta_j^1\rangle \in p.\mathcal{S}_{\text{rcv}} \wedge \langle \delta_j^1, \delta_j^2\rangle \in p.\mathcal{S}_{\text{int}}^* \wedge \langle \delta_j^2, \delta_j^3, \mathsf{ms}_j\rangle \in p.\mathcal{S}_{\text{snd}}$, $\forall j \in I, \sigma'(m.to)[j] = \delta_j^3$.

In essence, the rule OWStepReceiveT blends several steps of the widening semantics together for a single message: (a) it first receives the message and replicates it for all slots at a destination node; (b) performs receive-steps for the message's replicas at each slot; (c) takes a number of internal steps, allowed by the protocol's $\mathcal{S}_{\text{int}}$; and (d) takes a send-transition, eventually sending all emitted message, instrumented with the corresponding slots.

$$\textsc{BStepRecvB}\quad \frac{\begin{array}{c} m \in M \quad m.active \quad m.to \in \mathsf{dom}(\sigma) \\ \langle \sigma', \mathsf{ms} \rangle = \mathsf{receiveAndAct}(\sigma, n, m) \\ M' = M \setminus \{m\} \cup \{m[active \mapsto \mathsf{False}]\} \\ m' = bunch(\mathsf{ms}, m.to, m.from) \end{array}}{\langle \sigma, M \rangle \xRightarrow[\text{rcv}]{B} \langle \sigma', M' \cup \{m'\} \rangle}$$

$$\textsc{BStepRecvU}\quad \frac{m \in M \quad m.active \quad m.to \in \mathsf{dom}(\sigma) \quad m.msgs = \mathsf{ms} \quad M' = M \setminus \{m\} \cup \mathsf{ms}}{\langle \sigma, M \rangle \xRightarrow[\text{rcv}]{B} \langle \sigma, M' \rangle}$$

where $bunch(\mathsf{ms}, n_1, n_2) = \{msgs = \mathsf{ms}; from = n_1; to = n_2; active = \mathsf{True}\}$.

Figure 15. Added rules of the Bunching Semantics

Example 6. Continuing Example 5, with the same parameters, the optimising semantics will execute the transitions of an acceptor, *for all slots*, triggered by receiving a single `[RE, k]` message for a particular slot, sending back *all* the results for all the slots, which might either agree to accept the value or reject it.

The following lemma relates the optimising and the widening semantics.

Lemma 4 (Refinement for OW semantics). *For any $b \in \mathcal{B}_{\xRightarrow{\nabla^*}}$ there exists $b' \in \mathcal{B}_{\xRightarrow{\nabla}}$, such that b can be obtained from b' by replacing sequences of configurations $[\langle \sigma_k, M_k \rangle, \ldots, \langle \sigma_{k+m}, M_{k+m} \rangle]$ that have just a single node n, whose local state is affected in $\sigma_k, \ldots, \sigma_{k+m}$, by $[\langle \sigma_k, M_k \rangle, \langle \sigma_{k+m}, M_{k+m} \rangle]$.*

That is, behaviours in the optimised semantics are the same as in the widening semantics, modulo some sequences of locally taken steps that are being "compressed" to just the initial and the final configurations.

5.6 Bunching Semantics.

As the last step towards Multi-Paxos, we introduce the final network semantics that optimises executions according to $\xRightarrow{\nabla^*}$ described in previous section even further by making a simple addition to the message vocabulary of a slot-replicated SD Paxos—*bunched messages*. A bunched message simply packages together several messages, obtained typically as a result of a "compressed" execution via the optimised semantics from Section 5.5. We define two new rules for packaging and "unpackaging" certain messages in Figure 15. The two new rules can be added to enhance either of the versions of the slot-replicating semantics shown before. In essence, the only effect they have is to combine the messages resulting in the execution of the corresponding steps of an optimised widening (via BStepRecvB), and to unpackage the messages ms from a bunching message, adding them back to the soup (BStepRecvU). The following natural refinement result holds:

Lemma 5. *For any $b \in \mathcal{B}_{\xRightarrow{B}}$ there exists $b' \in \mathcal{B}_{\xRightarrow{\nabla^*}}$, such that b' can be obtained from b by replacing all bunched messages in b by their msgs-component.*

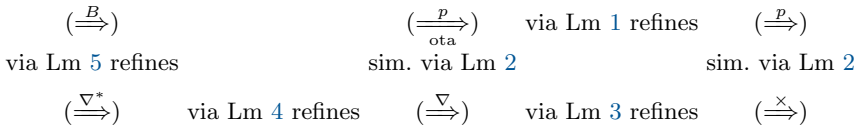


Figure 16. Refinement between different network semantics.

```
1  proposeM(val^ v, val v0) {
2    ⟨ assume(!(v0 = undef));
3      if (*v = undef) { *v := v0; }
4      return *v; ⟩ }
5  val vM[1..∞] := undef;
6  getR(int s) { return &(vM[s]); }
7  proposeM(getR(1), v);
8  proposeM(getR(2), v);
```

Figure 17. Specification of *Multi-Paxos* and interaction via a *register provider*.

The rule BStepRecvU enables effective local caching of the bunched messages, so they are processed *on demand* on the recipient side (*i.e.*, by the per-slot proposers), allowing the implementation to *skip* an entire round of Phase 1.

5.7 The Big Picture.

What exactly have we achieved by introducing the described above family of semantics? As illustrated in Figure 16, all behaviours of the leftmost-topmost, bunching semantics, which corresponds precisely to an implementation of Multi-Paxos with an "amortised" Phase 1, can be transitively related to the corresponding behaviours in the rightmost, vanilla slot-replicated version of a simple semantics (via the correspondence from Lemma 1) by constructing the corresponding refinement mappings [1], delivered by the proofs of Lemmas 3–5.

From the perspective of Rely/Guarantee reasoning, which was employed in Section 4, the refinement result from Figure 16 justifies the replacement of a semantics on the right of the diagram by one to the left of it, as all program-level assertions will remain substantiated by the corresponding system configurations, as long as they are *stable* (*i.e.*, resilient *wrt.* transitions taken by nodes different from the one being verified), which they are in our case.

6 Putting It All Together

We culminate our story of faithfully deconstructing and abstracting Paxos via a round-based register, as well as recasting Multi-Paxos via a series of network transformations, by showing how to *implement* the register-based abstraction from Section 3 in tandem with the network semantics from Section 5 in order to deliver provably correct, yet efficient, implementation of Multi-Paxos.

The crux of the composition of the two results—a register-based abstraction of SD Paxos and a family of semantics-preserving network transformations—is a convenient interface for the end client, so she could interact with a consensus instance via the `proposeM` method in lines 1–4 of Figure 17, no matter with which particular slot of a Multi-Paxos implementation she is interacting. To do so, we propose to introduce a *register provider*—a service that would give a client

a "reference" to the consensus object to interact with. Lines 6–7 of Figure 17 illustrate the interaction with the service provider, where the client requests two specific slots, 1 and 2, of Multi-Paxos by invoking `getR` and providing a slot parameter. In both cases the client proposes the very same value `v` in the two instances that run the same machinery. (Notice that, except for the reference to the consensus object, `proposeM` is identical to the `proposeP` on the right of Figure 2, which we have verified *wrt.* linearisability in Section 3.)

The implementation of Multi-Paxos that we have in mind resembles the one in Figures 3, 4 and 5 of Section 3, but where all the global data is provided by the register provider and passed by reference. What differs in this implementation with respect to the one in Section 3 and is hidden from the client is the semantics of the network layer used by the bottom layer (*cf.* left part of Figure 2) of the register-based implementation. The Multi-Paxos instances run (without changing the register's code) over this network layer, which "overloads" the meaning of the `send/receive` primitives from Figures 3 and 4 to follow the bunching network semantics, described in Section 5.6.

Theorem 4. *The implementation of Multi-Paxos that uses a register provider and bunching network semantics refines the specification in Figure 17.*

We implemented the register/network semantics in a proof-of-concept prototype written in Scala/Akka.[6] We relied on the abstraction mechanisms of Scala, allowing us to implement the register logic, verified in Section 4, separately from the network middle-ware, which has provided a family of Semantics from Section 5. Together, they provide a family of provably correct, modularly verified *distributed* implementations, coming with a simple *shared memory-like* interface.

7 Related Work

Proofs of Linearisability via Rely/Guarantee. Our work builds on the results of Boichat *et al.* [3], who were first to propose to a systematic deconstruction of Paxos into read/write operations of a *round-based register* abstraction. We extend and harness those abstractions, by intentionally introducing more non-determinism into them, which allows us to provide the first modular (*i.e.*, mutually independent) proofs of Proposer and Acceptor using Rely/Guarantee with linearisation points and prophecies. While several logics have been proposed recently to prove linearisability of concurrent implementations using Rely/Guarantee reasoning [14,18,19,26], none of them considers message-passing distributed systems or consensus protocols.

Verification of Paxos-family Algorithms. Formal verification of different versions of Paxos-family protocols *wrt.* inductive invariants and liveness has been a focus of multiple verification efforts in the past fifteen years. To name just a few, Lamport has specified and verified Fast Paxos [17] using TLA+ and its accompanying model checker [32]. Chand *et al.* used TLA+ to specify and verify Multi-Paxos implementation, similar to the one we considered in this work [5].

[6] The code is available at `https://github.com/certichain/protocol-combinators`.

A version of SD-Paxos has been verified by Kellomaki using the PVS theorem prover [13]. Jaskelioff and Merz have verified Disk Paxos in Isabelle/HOL [12]. More recently, Rahli *et al.* formalised an executable version of Multi-Paxos in EventML [24], a dialect of NuPRL. Dragoi *et al.* [8] implemented and verified SD-Paxos in the PSync framework, which implements a partially synchronised model [7], supporting automated proofs of system invariants. Padon *et al.* have proved the system invariants and the consensus property of both simple Paxos and Multi-Paxos using the verification tool Ivy [22,23].

Unlike all those verification efforts that consider (Multi-/Disk/Fast/...)Paxos as a *single monolithic protocol*, our approach provides the first *modular* verification of single-decree Paxos using Rely/Guarantee framework, as well as the first verification of Multi-Paxos that directly reuses the proof of SD-Paxos.

Compositional Reasoning about Distributed Systems. Several recent works have partially addressed modular formal verification of distributed systems. The IronFleet framework by Hawblitzel *et al.* has been used to verify both safety and liveness of a real-world implementation of a Paxos-based replicated state machine library and a lease-based shared key-value store [10]. While the proof is structured in a modular way by composing specifications in a way similar to our decomposition in Sections 3–4, that work does not address the linearisability and does not provide composition of proofs about complex protocols (*e.g.*, Multi-Paxos) from proofs about its subparts

The Verdi framework for deductive verification of distributed systems [29,31] suggests the idea of *Verified System Transformers* (VSTs), as a way to provide *vertical composition* of distributed system implementation. While Verdi's VSTs are similar in its purpose and idea to our network transformations, they *do not* exploit the properties of the protocol, which was crucial for us to verify Multi-Paxos's implementation.

The Disel framework [25,28] addresses the problem of *horizontal composition* of distributed protocols and their client applications. While we do not compose Paxos with any clients in this work, we believe its register-based specification could be directly employed for verifying applications that use Paxos as its subcomponent, which is what is demonstrated by our prototype implementation.

8 Conclusion and Future Work

We have proposed and explored two complementary mechanisms for modular verification of Paxos-family consensus protocols [15]: (a) non-deterministic register-based specifications in the style of Boichat *et al.* [3], which allow one to decompose the proof of protocol's linearisability into separate independent "layers", and (b) a family of protocol-aware transformations of network semantics, making it possible to reuse the verification efforts. We believe that the applicability of these mechanisms spreads beyond reasoning about Paxos and its variants and that they can be used for verifying other consensus protocols, such as Raft [21] and PBFT [4]. We are also going to employ network transformations to verify implementations of Mencius [20], and accommodate more protocol-specific optimisations, such as implementation of master leases and epoch numbering [6].

Acknowledgements. We thank the ESOP 2018 reviewers for their feedback. We also thank Amal Ahmed for her efforts as ESOP 2018 Programme Chair. The work by García-Pérez, Gotsman, and Meshman was supported by ERC Starting Grant H2020-EU 714729. Sergey's work was supported by EPSRC First Grant EP/P009271/1 "Program Logics for Compositional Specification and Verification of Distributed Systems".

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. In *LICS*, pages 165–175. IEEE Computer Society, 1988.
2. R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing Paxos, 2001. OAI-PMH server at infoscience.epfl.ch, record 52373 (http://infoscience.epfl.ch/record/52373).
3. R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing paxos. *SIGACT News*, 34(1):47–67, 2003.
4. M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *OSDI*, pages 173–186. USENIX Association, 1999.
5. S. Chand, Y. A. Liu, and S. D. Stoller. Formal Verification of Multi-Paxos for Distributed Consensus. In *FM*, volume 9995 of *LNCS*, pages 119–136, 2016.
6. T. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC*, pages 398–407. ACM, 2007.
7. B. Charron-Bost and S. Merz. Formal verification of a consensus algorithm in the heard-of model. *Int. J. Software and Informatics*, 3(2-3):273–303, 2009.
8. C. Dragoi, T. A. Henzinger, and D. Zufferey. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*, pages 400–415. ACM, 2016.
9. I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.
10. C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. IronFleet: proving practical distributed systems correct. In *SOSP*, pages 1–17. ACM, 2015.
11. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
12. M. Jaskelioff and S. Merz. Proving the correctness of disk paxos. *Archive of Formal Proofs*, 2005, 2005.
13. P. Kellomäki. An Annotated Specification of the Consensus Protocol of Paxos Using Superposition in PVS. Technical Report Report 36, Tampere University of Technology. Institute of Software Systems, 2004.
14. A. Khyzha, A. Gotsman, and M. J. Parkinson. A Generic Logic for Proving Linearizability. In *FM*, volume 9995 of *LNCS*, pages 426–443. Springer, 2016.
15. L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
16. L. Lamport. Paxos made simple. *SIGACT News*, 32, 2001.
17. L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.
18. H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, pages 459–470. ACM, 2013.
19. H. Liang and X. Feng. A program logic for concurrent objects under fair scheduling. In *POPL*, pages 385–399. ACM, 2016.

20. Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building Efficient Replicated State Machine for WANs. In *OSDI*, pages 369–384. USENIX Association, 2008.
21. D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference*, pages 305–319, 2014.
22. O. Padon, G. Losa, M. Sagiv, and S. Shoham. Paxos made EPR: decidable reasoning about distributed protocols. *PACMPL*, 1(OOPSLA):108:1–108:31, 2017.
23. O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham. Ivy: safety verification by interactive generalization. In *PLDI*, pages 614–630. ACM, 2016.
24. V. Rahli, D. Guaspari, M. Bickford, and R. L. Constable. Formal specification, verification, and implementation of fault-tolerant systems using EventML. In *AVOCS*. EASST, 2015.
25. I. Sergey, J. R. Wilcox, and Z. Tatlock. Programming and proving with distributed protocols. *PACMPL*, 2(POPL):28:1–28:30, 2018.
26. V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.
27. R. van Renesse and D. Altinbuken. Paxos Made Moderately Complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, 2015.
28. J. R. Wilcox, I. Sergey, and Z. Tatlock. Programming Language Abstractions for Modularly Verified Distributed Systems. In *SNAPL*, volume 71 of *LIPIcs*, pages 19:1–19:12. Schloss Dagstuhl, 2017.
29. J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, pages 357–368. ACM, 2015.
30. G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, Cambridge, Massachusetts, 1993.
31. D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. E. Anderson. Planning for change in a formal verification of the Raft Consensus Protocol. In *CPP*, pages 154–165. ACM, 2016.
32. Y. Yu, P. Manolios, and L. Lamport. Model Checking TLA$^+$ Specifications. In *CHARME*, volume 1703 of *LNCS*, pages 54–66. Springer, 1999.

A Proof Outline of Module *Paxos*

Proof (Theorem 1). By the following proof of linearisation. The following predicates state the relation that connects the concrete with the abstract state and the invariant of Paxos:

$$\begin{array}{l} \mathit{AbsP} \equiv \mathtt{abs\_vP} = \mathtt{vRC} \wedge (\mathtt{abs\_vP} = \mathtt{undef} \vee \mathtt{abs\_vP} \in \mathtt{valsRC}) \\ \mathit{InvP} \equiv \mathtt{valsRC} \subseteq \{v \mid \exists (i, b).\ \mathtt{ptp}[i] = (b,\ v)\}. \end{array}$$

We consider actions (`ProposeP1`)

$$\begin{array}{l} \mathtt{abs\_vP} = \mathtt{vP} = \mathtt{undef} \wedge v \neq \mathtt{undef} \wedge v \in \mathtt{valsRC} = V \wedge \\ I = \{i \mid \mathtt{ptp}[i] = (\mathtt{true},\ v)\} \wedge (\bigwedge_{i \in I} \mathtt{abs\_resP}[i] = \mathtt{undef}) \\ \rightsquigarrow \\ \mathtt{abs\_vP} = \mathtt{vP} = v \wedge \mathtt{valsRC} = V \wedge \\ (\bigwedge_{i \in I} (\mathtt{ptp}[i] = (\mathtt{false},\ v) \wedge \mathtt{abs\_resP}[i] = v)), \end{array}$$

(ProposeP2)

$$\begin{array}{l}\texttt{abs\_vP} = \texttt{vP} = v \wedge v \neq \texttt{undef} \wedge I = \{i \mid \texttt{ptp}[i] = (\texttt{true},\ v)\} \wedge \\ (\bigwedge_{i \in I} \texttt{abs\_resP}[i] = \texttt{undef}) \\ \rightsquigarrow \\ \texttt{abs\_vP} = \texttt{vP} = v \wedge (\bigwedge_{i \in I}(\texttt{ptp}[i] = (\texttt{false},\ v) \wedge \texttt{abs\_resP}[i] = v)),\end{array}$$

(ProposeP3)$_i$

$$\begin{array}{l}\texttt{abs\_vP} = \texttt{vP} = \texttt{undef} \wedge v \neq \texttt{undef} \wedge v = \texttt{ptp}[i] \wedge \\ I = \{i \mid \texttt{ptp}[i] = (\texttt{true},\ v)\} \wedge (\bigwedge_{i \in I} \texttt{abs\_resP}[i] = \texttt{undef}) \\ \rightsquigarrow \\ \texttt{abs\_vP} = \texttt{vP} = v \wedge (\bigwedge_{i \in I}(\texttt{ptp}[i] = (\texttt{false},\ v) \wedge \texttt{abs\_resP}[i] = v)),\end{array}$$

and (ProposeP4)$_i$

$$\begin{array}{l}\texttt{abs\_vP} = \texttt{vP} = v \wedge v \neq \texttt{undef} \wedge I = \{i \mid \texttt{ptp}[i] = (\texttt{true},\ v)\} \wedge \\ (\bigwedge_{i \in I} \texttt{abs\_resP}[i] = \texttt{undef}) \wedge v' \neq v \wedge v' \neq \texttt{undef} \wedge \\ \texttt{ptp}[i] = v' \wedge \texttt{abs\_resP}[i] = \texttt{undef} \\ \rightsquigarrow \\ \texttt{abs\_vP} = \texttt{vP} = v \wedge (\bigwedge_{i \in I}(\texttt{ptp}[i] = (\texttt{false},\ v) \wedge \texttt{abs\_resP}[i] = v)) \wedge \\ \texttt{ptp}[i] = \texttt{undef} \wedge \texttt{abs\_resP}[i] = v.\end{array}$$

The guarantee relation for `proposeP(v0)` is

$$(\texttt{ProposeP1}) \cup (\texttt{ProposeP2}) \cup (\texttt{ProposeP3})_{\texttt{pid()}} \cup (\texttt{ProposeP4})_{\texttt{pid()}},$$

where `pid()` is the process identifier of the proposer, and the rely relation is

$$(\texttt{ProposeP1}) \cup (\texttt{ProposeP2}) \cup \bigcup_{i \neq \texttt{pid()}} ((\texttt{ProposeP3})_i \cup (\texttt{ProposeP4})_i).$$

```
1  val abs_vP := undef;
2  (bool × val) ptp[1..n] := undef;
3  single bool abs_resP[1..n] := undef;
4  proposeP(val v0) {
5    int k; bool res; val v;
6    assume(!(v0 = undef));
7    { ptp[pid()] = undef ∧ AbsP ∧ InvP }
8    k := pid(); ptp[pid()] := (true, v0);
9    { pid() = k ∧ AbsP ∧ InvP }
10   do {
11     { pid() = ((k − 1) mod n) + 1 ∧ AbsP ∧ InvP }
12     ⟨ (res, v) := proposeRC(k, v0);
13       if (res) {
14         for (i := 1, i <= n, i++) {
15           if (ptp[i] = (true, v)) { lin(i); ptp[i] := (false, v); } }
16         if (!(v = v0)) { lin(pid()); ptp[pid()] := (false, v0); } }
17       { ((res = true ∧ abs_vP = abs_resP[pid()] = v) ∨ res = false) ∧
         { ptp[pid()] = (false, v) ∧ pid() = k mod n ∧ AbsP ∧ InvP }
```

```
18      ⟩
19      { ((res = true ∧ abs_vP = abs_resP[pid()] = v) ∨ res = false) ∧
        { ptp[pid()] = (false, v) ∧ pid() = k mod n ∧ AbsP ∧ InvP }
20      k := k + n;
21      { ((res = true ∧ abs_vP = abs_resP[pid()] = v) ∨ res = false) ∧
        { ptp[pid()] = (false, v) ∧ pid() = k mod n ∧ AbsP ∧ InvP }
22    } while (!res);
23    { abs_vP = abs_resP[pid()] = v ∧                                }
      { ptp[pid()] = (false, v) ∧ pid() = k mod n ∧ AbsP ∧ InvP       }
24    return v; }
```

□

B Proof Outline of Module *Round-Based Consensus*

Proof (Theorem 2). By the following proof of linearisation. The following predicates state the abstract relation *AbsRC* between the concrete and the abstract state in the instrumented implementation of Figure 8.

$$\begin{aligned} \mathit{AbsRC} \equiv\ & \texttt{abs\_vRC} = \texttt{vRR} \wedge \texttt{abs\_roundRC} \leq \texttt{roundRR} \wedge \\ & \texttt{abs\_valsRC} = \texttt{valsRR} \setminus \{\texttt{undef}\}. \end{aligned}$$

Variables `vRR`, `roundRR` and `valsRR` are respectively the decided value, the round and the set of values from the module *Round-Based Register*. Predicate *AbsRC* ensures that the abstract `abs_vRC` and concrete `vRR` coincide, that the abstract round `abs_roundRC` is less or equal than the concrete `roundRR`, and that the abstract `abs_valsRC` corresponds to the concrete `valsRR` minus `undef`.

The following predicate states the invariant *InvRC* of Round-Based Consensus.

$$\mathit{InvRC} \equiv (\texttt{abs\_vRC} = \texttt{undef} \vee (v \neq \texttt{undef} \wedge \texttt{abs\_vRC} = v \wedge v \in \texttt{abs\_valsRC})).$$

The invariant ensures that either no value has been decided yet (*i.e.* $\texttt{abs\_vRC} = \texttt{undef}$), or otherwise a value $v \neq \texttt{undef}$ has been decided (*i.e.* $\texttt{abs\_vRC} = v$) and the abstract `abs_valsRC` contains v.

Now we define the rely and guarantee relations. We consider the actions $(\texttt{ProposeRC1})_k$

$$\begin{aligned} & \texttt{abs\_roundRC} \leq k = \texttt{roundRR} \wedge \texttt{abs\_vRC} = \texttt{undef} \wedge \texttt{abs\_valsRC} = V \\ & \rightsquigarrow \\ & \texttt{abs\_roundRC} = \texttt{roundRR} = k \wedge \texttt{abs\_vRC} = v \wedge v \neq \texttt{undef} \wedge \\ & \texttt{abs\_valsRC} = V \cup \{v\} \wedge \texttt{abs\_resRC} = (\texttt{true}, v), \end{aligned}$$

$(\texttt{ProposeRC2})_k$

$$\begin{aligned} & \texttt{abs\_roundRC} \leq k = \texttt{roundRR} \wedge \texttt{abs\_vRC} = v \wedge v \neq \texttt{undef} \wedge \\ & v \in \texttt{abs\_valsRC} = V \\ & \rightsquigarrow \\ & \texttt{abs\_roundRC} = \texttt{roundRR} = k \wedge \texttt{abs\_vRC} = v \wedge v \neq \texttt{undef} \wedge \texttt{abs\_valsRC} = V, \end{aligned}$$

$(\texttt{ProposeRC3})_k$

$$\begin{array}{l}\texttt{abs\_roundRC} = k \wedge \texttt{roundRR} = k' \wedge k \leq k' \wedge \texttt{abs\_vRC} = \texttt{undef} \wedge \\ \texttt{abs\_valsRC} = V \\ \rightsquigarrow \\ \texttt{abs\_roundRC} = k \wedge \texttt{roundRR} = k' \wedge \texttt{abs\_vRC} = \texttt{undef} \wedge v \neq \texttt{undef} \wedge \\ \texttt{abs\_valsRC} = V \cup \{v\},\end{array}$$

and $(\texttt{ReadRC})_k$

$$\texttt{roundRR} \leq k \rightsquigarrow \texttt{roundRR} = k.$$

The guarantee relation for `proposeRC(k, v0)` is

$$(\texttt{ProposeRC1})_k \cup (\texttt{ProposeRC2})_k \cup (\texttt{ProposeRC3})_k \cup (\texttt{ReadRC})_k,$$

and the rely relation is

$$\bigcup_{(k \bmod n) \neq (k \bmod n)} ((\texttt{ProposeRC1})_k \cup (\texttt{ProposeRC2})_k \cup (\texttt{ProposeRC3})_k \cup (\texttt{Read})_k).$$

The proof outline below helps to show that if $AbsRel \wedge Inv$ holds at the beginning of the method invocation of the annotated program, then it also holds at the end of the method invocation after the abstract operation has been performed at the linearisation point, and that the abstract result `abs_resRC` coincides with the result of the concrete method. It also helps to show that the method ensures the guarantee relation, that is, the states between each atomic operation are related by the guarantee condition.

```
1  val abs_vRC := undef;
2  int abs_roundRC := 0;
3  set of val abs_valsRC := {};
4  proposeRC(int k, val v0) {
5    single (bool × val) abs_resRC := undef;
6    bool res; val v;
7    assume(!(v0 = undef));
8    assume(pid() = ((k - 1) mod n) + 1;
9    { pid() = ((k − 1) mod n) + 1 ∧ v0 ≠ undef ∧ abs_resRC = undef ∧
       AbsRC ∧ InvRC }
10   ⟨ (res, v) := read(k);
11     if (res = false) { linRC(undef, _); }
12     { ((res = true ∧ v = vRR ∧ vRR ≠ undef ∧ k = roundRR)
         ∨ (resR = true ∧ v ∈ valsRR)
         ∨ (res = false ∧ abs_resRC = (false, _))) ∧
       pid() = ((k − 1) mod n) + 1 ∧ v0 ≠ undef ∧ AbsRC ∧ InvRC }
13   ⟩
14   { ((res = true ∧ v = vRR ∧ vRR ≠ undef ∧ k ≤ roundRR)
       ∨ (res = true ∧ v ∈ valsRR)
       ∨ (res = false ∧ abs_resRC = (false, _))) ∧
     pid() = ((k − 1) mod n) + 1 ∧ v0 ≠ undef ∧ AbsRC ∧ InvRC }
```

```
15  if (res) {
16    {((v = vRR ∧ vRR ≠ undef ∧ k ≤ roundRR) ∨ (v ∈ valsRR)) ∧
      pid() = ((k − 1) mod n) + 1 ∧ v0 ≠ undef ∧ AbsRC ∧ InvRC}
17    if (v = undef) {
18      {v = undef ∧ v ∈ valsRR ∧
        pid() = ((k − 1) mod n) + 1 ∧ v0 ≠ undef ∧ AbsRC ∧ InvRC}
19      v := v0;
20      {v = v0 ∧
        pid() = ((k − 1) mod n) + 1 ∧ v0 ≠ undef ∧ AbsRC ∧ InvRC}
21    }
22    {((v = vRR ∧ k ≤ roundRR) ∨ v ∈ valsRR ∨ v = v0) ∧
      pid() = ((k − 1) mod n) + 1 ∧ v ≠ undef ∧ AbsRC ∧ InvRC}
23    ⟨ res := write(k, v);
24      if (res) { linRC(v, true); }
25      else { linRC(v, false); }
26      {((res = true ∧ vRR = v ∧ k = abs_roundRC = roundRR ∧
          abs_resRC = (true, v))
        ∨ (res = false ∧ abs_resRC = false)) ∧
        v ∈ valsRC ∧ pid() = ((k − 1) mod n) + 1 ∧ AbsRC ∧ InvRC}
27      ⟩
28    {((res = true ∧ vRR = v ∧ k ≤ roundRR ∧ abs_resRC = (true, v))
      ∨ (res = false ∧ abs_resRC = (false, _))) ∧
      v ∈ valsRC ∧ pid() = ((k − 1) mod n) + 1 ∧ AbsRC ∧ InvRC}
29    if (res) {
30      {vRR = v ∧ k ≤ roundRR ∧ abs_resRC = (true, v) ∧
        v ∈ valsRC ∧ pid() = ((k − 1) mod n) + 1 ∧ AbsRC ∧ InvRC}
31      return (true, v); } }
32  {abs_resRC = (false, _) ∧ pid() = ((k − 1) mod n) + 1 ∧ AbsRC ∧ InvRC}
33  return (false, _); }
```

□

C Proof Outline of Module *Round-Based Register*

Proof (Theorem 3). We use the predicates *sent* and *received* to represent the state of the network. The predicate $sent(i, j, m)$ is true iff process i sent message m to process j. The predicate $received(j, i, m)$ is true iff, in turn, process j received message m from i.

We introduce the following abbreviations for the state of the network:

$$
\begin{array}{rcl}
sent(i,j,msg,m) & \equiv & m \in M \wedge m.from = i \wedge m.to = j \wedge \\
 & & m.content = msg \\
received(j,i,msg,m) & \equiv & m \in M \wedge m.from = i \wedge m.to = j \wedge \\
 & & m.content = msg \wedge m.active = \mathsf{False} \\
reqRE(i,j,k,m) & \equiv & sent(i,j,\texttt{[RE, } k\texttt{]},m) \wedge \\
 & & \neg(\exists(k',v).\ sent(j,i,\texttt{[ackRE, } k\texttt{, } v\texttt{, } k'\texttt{]},m)) \\
ackRE(j,i,k,v,k',m) & \equiv & received(j,i,\texttt{[RE, } k\texttt{]},m) \wedge \\
 & & sent(j,i,\texttt{[ackRE, } k\texttt{, } v\texttt{, } k'\texttt{]},m) \\
reqWR(i,j,k,v,m) & \equiv & sent(i,j,\texttt{[WR, } k\texttt{, } v\texttt{])},m) \wedge \\
 & & \neg sent(j,i,\texttt{[ackWR, } k\texttt{]},m) \\
ackWR(j,i,k,v) & \equiv & received(j,i,\texttt{[WR, } k\texttt{, } v\texttt{]},m) \wedge \\
 & & sent(j,i,\texttt{[ackWR, } k\texttt{]},m)
\end{array}
$$

In our proofs of linearisation we consider the following proof rules

$$
\frac{}{\begin{array}{c}\{\neg(\exists m'.\ m' = m \wedge sent(\texttt{pid()},j,msg,m')) \wedge p\} \\ \texttt{send(}j\texttt{, } msg\texttt{)} \\ \{sent(\texttt{pid()},j,msg,m) \wedge p\}\end{array}}
$$

$$
\frac{}{\begin{array}{c}\{\neg(\exists m'.\ m' = m \wedge received(\texttt{pid()},i,msg,m')) \wedge p\} \\ \texttt{(i, msg) := receive()} \\ \{sent(i,\texttt{pid()},msg,m) \wedge received(\texttt{pid()},i,msg,m) \wedge \texttt{i} = i \wedge \texttt{msg} = msg \wedge p\}\end{array}},
$$

which are sound under the network semantics of Section 5 and under the operational semantics of our pseudo-code that we outline in Appendix D.

The following abbreviations express the invariant that connects the auxiliary variables `count_r` and `count_w` with the cardinality of the corresponding quorums.

$$
\begin{array}{rcl}
CountR(k) & \equiv & \texttt{count\_r[}k\texttt{]} = \\
 & & |\{j \mid \exists(k',v.m).\ ackRE(j,((k-1) \bmod n)+1,k,v,k',m)\}| \\
CountW(k) & \equiv & \texttt{count\_w[}k\texttt{]} = \\
 & & |\{j \mid \exists(v,m).\ ackWR(j,((k-1) \bmod n)+1,k,v,m)\}| \\
Count(k) & \equiv & CountR(k) \wedge CountW(k)
\end{array}
$$

The following predicates state the abstract relation $AbsRR$ between the concrete and the abstract state in the instrumented implementation of Figures 10 and 11.

$$
\begin{aligned}
\mathit{AbsV}(v,k) &\equiv (\texttt{abs\_vRR} = v \neq \texttt{undef} \wedge k = \texttt{abs\_roundRR}) \\
&\quad \Longrightarrow \exists Q.\ (|Q| \geq \lceil (n+1)/2 \rceil \wedge \\
&\qquad \forall j \in Q.\ \exists m.\ \mathit{ackWR}(j, ((k-1) \bmod n) + 1, k, v, m)) \\
\mathit{AbsVals}(v) &\equiv v \in \texttt{abs\_valsRR} \\
&\quad \Longrightarrow (v = \texttt{undef} \\
&\qquad \vee \exists (j,k,m).\ \mathit{reqWR}(((k-1) \bmod n) + 1, j, k, v, m)) \\
\mathit{AbsRound}(k) &\equiv \texttt{abs\_roundRR} = k = 0 \\
&\quad \vee (\texttt{abs\_roundRR} = \max\{k \mid \texttt{count\_w[}k\texttt{]} \geq \lceil (n+1)/2 \rceil \\
&\qquad \vee \texttt{count\_r[}k\texttt{]} \geq \lceil (n+1)/2 \rceil\}) \\
\mathit{AbsRR} &\equiv \forall (v,k).\ (\mathit{AbsV}(v,k) \wedge \mathit{AbsVals}(v) \wedge \mathit{AbsRound}(k))
\end{aligned}
$$

The following predicates state the invariant *InvRR* of *Round-Based Register*.

$$
\begin{aligned}
\mathit{Read}(j,k) &\equiv j\texttt{.r} = k > 0 \\
&\quad \Longrightarrow (\exists (v,m).\ \mathit{ackWR}(j, ((k-1) \bmod n) + 1, k, v, m) \\
&\qquad \vee \exists (k', m).\ \mathit{ackRE}(j, ((k-1) \bmod n) + 1, k, v, k', m)) \\
\mathit{Val}(j,v) &\equiv j\texttt{.v} = v \neq \texttt{undef} \\
&\quad \Longleftrightarrow \exists (k,m).\ \mathit{ackWR}(j, ((k-1) \bmod n) + 1, k, v, m) \\
\mathit{ProphR1}(k) &\equiv (\texttt{proph\_r[}k\texttt{]} = \texttt{(}k'\texttt{, } v\texttt{)} \wedge \texttt{count\_r[}k\texttt{]} \geq \lceil (n+1)/2 \rceil) \\
&\quad \Longrightarrow \texttt{abs\_res\_r[}k\texttt{]} = (\texttt{true}, v) \\
\mathit{ProphR2}(k) &\equiv \texttt{proph\_r[}k\texttt{]} = \texttt{(false, \_)} \Longrightarrow \texttt{abs\_res\_r[}k\texttt{]} = \texttt{undef} \\
\mathit{ProphW1}(k) &\equiv (\texttt{proph\_w[}k\texttt{]} = \texttt{true} \wedge \texttt{count\_w[}k\texttt{]} \geq \lceil (n+1)/2 \rceil) \\
&\quad \Longrightarrow \texttt{abs\_res\_w[}k\texttt{]} = \texttt{true} \\
\mathit{ProphW2}(k) &\equiv (\texttt{proph\_w[}k\texttt{]} = \texttt{false} \wedge \texttt{count\_w[}k\texttt{]} > 0) \\
&\quad \Longrightarrow \texttt{abs\_res\_w[}k\texttt{]} = \texttt{false} \\
\mathit{ProphW3}(k) &\equiv (\texttt{proph\_w[}k\texttt{]} = \texttt{false} \wedge \texttt{count\_w[}k\texttt{]} = 0) \\
&\quad \Longrightarrow \texttt{abs\_res\_w[}k\texttt{]} = \texttt{undef} \\
\mathit{Proph}(k) &\equiv \mathit{ProphR1}(k) \wedge \mathit{ProphR2}(k) \wedge \\
&\quad \mathit{ProphW1}(k) \wedge \mathit{ProphW2}(k) \wedge \mathit{ProphW3}(k) \\
\mathit{InvRR} &\equiv \forall (j,k,v).\ (j\texttt{.r} \geq j\texttt{.w} \wedge \mathit{Read}(j,k) \wedge \\
&\qquad \mathit{Val}(j,v) \wedge \mathit{Count}(k) \wedge \mathit{Proph}(k)).
\end{aligned}
$$

The proof of Theorem 3 involves two proofs of linearisation for `read` and `write` respectively, and one proof proving that the code of acceptors meets the invariant $\mathit{AbsRR} \wedge \mathit{InvRR}$.

Now we define the rely and guarantee relations. We consider three kinds of actors in the system corresponding to each of the proofs: reader, writer, and acceptor. We define first the guarantee relations for each of the actors, and then we express the rely relation for each actor as a combination of the guarantee relations of the other actors. Consider the actions $(\texttt{Send})_{(i,j,msg)}$

$$m.\mathit{from} = i \wedge m.\mathit{to} = j \wedge m.\mathit{content} = msg \rightsquigarrow \mathit{sent}(i, j, msg, m),$$

and $(\texttt{Receive})_{(j,i,msg)}$

$$
\begin{aligned}
&m.\mathit{from} = i \wedge m.\mathit{to} = j \wedge m.\mathit{content} = msg \wedge \neg(\mathit{received}(j, i, msg, m)) \\
&\rightsquigarrow \mathit{sent}(i, j, msg, m) \wedge \mathit{received}(j, i, msg, m),
\end{aligned}
$$

which model sending and receiving a message.

A reader can also perform actions $(\texttt{ReadFails1})_k$

$$\begin{array}{l} \mathit{received}(i, j, [\texttt{nackRE'}, k]), m) \wedge i = ((k-1) \bmod n) + 1 \wedge \\ \texttt{proph\_r}[k] = (\texttt{false}, \_) \wedge \texttt{abs\_res\_r}[k] = \texttt{undef} \wedge k \geq \texttt{abs\_roundRR} \\ \rightsquigarrow \\ \mathit{received}(i, j, [\texttt{nackRE'}, k], m) \wedge \texttt{proph\_r}[k] = \texttt{undef} \wedge \\ \texttt{abs\_res\_r}[k] = (\texttt{false}, \_) \wedge \texttt{abs\_roundRR} = k \end{array}$$

and $(\texttt{ReadFails2})_k$

$$\begin{array}{l} \mathit{received}(i, j, [\texttt{nackRE}, k], m) \wedge i = ((k-1) \bmod n) + 1 \wedge \\ \texttt{proph\_r}[k] = (\texttt{false}, \_) \wedge \texttt{abs\_res\_r}[k] = \texttt{undef} \wedge k < \texttt{abs\_roundRR} = k' \\ \rightsquigarrow \\ \mathit{received}(i, j, [\texttt{nackRE}, k], m) \wedge \texttt{proph\_r}[k] = \texttt{undef} \wedge \\ \texttt{abs\_res\_r}[k] = (\texttt{false}, \_) \wedge \texttt{abs\_roundRR} = k'. \end{array}$$

The guarantee relation of $\texttt{read}(k)$ is the one induced by the union of these actions as follows:

$$\begin{array}{rl} (\texttt{Reader})_k \equiv & \bigcup_j (\texttt{Send})_{((k \bmod n)+1, j, [\texttt{RE}, k])} \cup \\ & \bigcup_{j,v,k'} ((\texttt{Receive})_{((k \bmod n)+1, j, [\texttt{ackRE}, k, v, k'])} \cup \\ & \qquad (\texttt{Receive})_{((k \bmod n)+1, j, [\texttt{nackRE}, k])}) \cup \\ & (\texttt{ReadFails1})_k \cup (\texttt{ReadFails2})_k. \end{array}$$

Now we focus on a writer, which, additionally to sending reads and receiving acknowledgements, can perform action $(\texttt{WriteFails1})_{(k,v)}$

$$\begin{array}{l} \mathit{received}(i, j, [\texttt{nackWR}, k], m) \wedge i = ((k-1) \bmod n) + 1 \wedge v \neq \texttt{undef} \wedge \\ \texttt{proph\_w}[k] = \texttt{false} \wedge \texttt{abs\_valsRR} = V \wedge \texttt{count\_w}[k] = 0 \wedge \\ \texttt{abs\_res\_w}[k] = \texttt{undef} \\ \rightsquigarrow \\ \mathit{received}(i, j, [\texttt{nackWR}, k], m) \wedge i = (k \bmod n) + 1 \wedge \\ \texttt{proph\_w}[k] = \texttt{undef} \wedge \texttt{abs\_valsRR} = V \cup \{v\} \wedge \texttt{count\_w}[k] = 0 \wedge \\ \texttt{abs\_res\_w}[k] = \texttt{false}. \end{array}$$

The guarantee relation of $\texttt{write}(k, v)$ is the one induced by the union of these actions as follows:

$$\begin{array}{rl} (\texttt{Writer})_{(k,v)} \equiv & \bigcup_j (\texttt{Send})_{(((k-1) \bmod n)+1, j, [\texttt{WR}, k, v])} \cup \\ & \bigcup_j ((\texttt{Receive})_{(((k-1) \bmod n)+1, j, [\texttt{ackWR}, k])} \cup \\ & \qquad (\texttt{Receive})_{(((k-1) \bmod n)+1, j, [\texttt{nackWR}, k])}) \cup \\ & (\texttt{WriteFails1})_{(k,v)}. \end{array}$$

Now to the acceptors, which can receive read and write requests and send non-acknowledgements to them. They can also perform actions $(\texttt{ReadSucceeds1})_j$

$$
\begin{array}{l}
\mathit{reqRE}(i,j,k,m) \wedge i = ((k-1) \bmod n) + 1 \wedge k \geq j.\texttt{r} \wedge \\
\texttt{abs\_res\_r}[k] = \texttt{undef} \wedge \texttt{proph\_r}[k] = \texttt{(true, } v\texttt{)} \wedge v \in \texttt{abs\_valsRR} = V \wedge \\
\texttt{count\_r}[k] = c = \lceil (n+1)/2 \rceil - 1 \wedge k \geq \texttt{abs\_roundRR} \\
\rightsquigarrow \\
\mathit{ackRE}(j,i,k,j.\texttt{v}, j.\texttt{w}, m) \wedge j.\texttt{r} = k \wedge \texttt{count\_r}[k] = c+1 \wedge \\
\texttt{proph\_r}[k] = \texttt{(true, } v\texttt{)} \wedge \texttt{abs\_roundRR} = k \wedge \texttt{abs\_valsRR} = V \wedge \\
\texttt{abs\_res\_r}[k] = \texttt{(true, } v\texttt{)},
\end{array}
$$

$(\texttt{ReadSucceeds2})_j$

$$
\begin{array}{l}
\mathit{reqRE}(i,j,k,m) \wedge i = ((k-1) \bmod n) + 1 \wedge k \geq j.\texttt{r} \wedge \\
\texttt{abs\_res\_r}[k] = \texttt{undef} \wedge \texttt{proph\_r}[k] = \texttt{(true, } v\texttt{)} \wedge v \in \texttt{abs\_valsRR} = V \wedge \\
\texttt{count\_r}[k] = c = \lceil (n+1)/2 \rceil - 1 \wedge k < \texttt{abs\_roundRR} = r \\
\rightsquigarrow \\
\mathit{ackRE}(j,i,k,j.\texttt{v}, j.\texttt{w}, m) \wedge j.\texttt{r} = k \wedge \texttt{count\_r}[k] = c+1 \wedge \\
\texttt{proph\_r}[k] = \texttt{(true, } v\texttt{)} \wedge \texttt{abs\_roundRR} = r \wedge \texttt{abs\_valsRR} = V \wedge \\
\texttt{abs\_res\_r}[k] = \texttt{(true, } v\texttt{)},
\end{array}
$$

$(\texttt{AckRead1})_j$

$$
\begin{array}{l}
\mathit{reqRE}(i,j,k,m) \wedge i = ((k-1) \bmod n) + 1 \wedge k \geq j.\texttt{r} \wedge \\
\texttt{count\_r}[k] = c \neq \lceil (n+1)/2 \rceil - 1 \\
\rightsquigarrow \\
\mathit{ackRE}(j,i,k,j.\texttt{v}, j.\texttt{w}, m) \wedge j.\texttt{r} = k \wedge \texttt{count\_r}[k] = c+1,
\end{array}
$$

$(\texttt{AckRead2})_j$

$$
\begin{array}{l}
\mathit{reqRE}(i,j,k,m) \wedge i = ((k-1) \bmod n) + 1 \wedge k \geq j.\texttt{r} \wedge \\
\texttt{count\_r}[k] = c \wedge \texttt{proph\_r}[k] = p \neq \texttt{(true, } v\texttt{)} \\
\rightsquigarrow \\
\mathit{ackRE}(j,i,k,j.\texttt{v}, j.\texttt{w}, m) \wedge j.\texttt{r} = k \wedge \texttt{count\_r}[k] = c+1 \wedge \\
\texttt{proph\_r}[k] = p,
\end{array}
$$

$(\texttt{AckRead3})_j$

$$
\begin{array}{l}
\mathit{reqRE}(i,j,k,m) \wedge i = ((k-1) \bmod n) + 1 \wedge k \geq j.\texttt{r} \wedge \\
\texttt{count\_r}[k] = c \wedge \texttt{abs\_res\_r}[k] = ar \neq \texttt{undef} \\
\rightsquigarrow \\
\mathit{ackRE}(j,i,k,j.\texttt{v}, j.\texttt{w}, m) \wedge j.\texttt{r} = k \wedge \texttt{count\_r}[k] = c+1 \wedge \\
\texttt{abs\_res\_r}[k] = ar,
\end{array}
$$

$(\texttt{WriteSucceeds})_j$

$$
\begin{array}{l}
\quad reqWR(i,j,k,v,m) \wedge i = ((k-1) \bmod n) + 1 \wedge v \neq \texttt{undef} \wedge k \geq j.\texttt{r} \wedge \\
\quad \texttt{abs\_valsRR} = V \wedge \texttt{abs\_res\_w}[k] = \texttt{undef} \wedge \texttt{proph\_w}[k] = \texttt{true} \wedge \\
\quad \texttt{count\_w}[k] = c = \lceil (n+1)/2 \rceil - 1 \wedge k \geq \texttt{abs\_roundRR} \\
\quad \rightsquigarrow \\
\quad ackWR(j,i,k,v,m) \wedge j.\texttt{r} = k \wedge j.\texttt{w} = k \wedge j.\texttt{v} = v \wedge \\
\quad \texttt{count\_w}[k] = c+1 \wedge \texttt{proph\_w}[k] = \texttt{true} \wedge \texttt{abs\_valsRR} = V \cup \{v\} \wedge \\
\quad \texttt{abs\_roundRR} = k \wedge \texttt{abs\_vRR} = v \wedge \texttt{abs\_res\_w}[k] = \texttt{true},
\end{array}
$$

$(\texttt{AckWrite1})_j$

$$
\begin{array}{l}
reqWR(i,j,k,v,m) \wedge i = ((k-1) \bmod n) + 1 \wedge v \neq \texttt{undef} \wedge k \geq j.\texttt{r} = r \wedge \\
\texttt{abs\_res\_w}[k] = \texttt{undef} \wedge \texttt{proph\_w}[k] = \texttt{true} \wedge \\
\texttt{count\_w}[k] = c \neq \lceil (n+1)/2 \rceil - 1 \\
\rightsquigarrow \\
ackWR(j,i,k,v,m) \wedge j.\texttt{r} = k \wedge j.\texttt{w} = k \wedge j.\texttt{v} = v \wedge \\
\texttt{count\_w}[k] = c+1 \wedge \texttt{proph\_w}[k] = \texttt{true} \wedge \texttt{abs\_res\_w}[k] = \texttt{undef},
\end{array}
$$

$(\texttt{WriteFails2})_j$

$$
\begin{array}{l}
reqWR(i,j,k,v,m) \wedge i = ((k-1) \bmod n) + 1 \wedge v \neq \texttt{undef} \wedge k \geq j.\texttt{r} = r \wedge \\
\texttt{abs\_res\_w}[k] = \texttt{undef} \wedge \texttt{proph\_w}[k] = \texttt{false} \wedge v \in \texttt{abs\_valsRR} = V \wedge \\
\texttt{count\_w}[k] = c \\
\rightsquigarrow \\
ackWR(j,i,k,v,m) \wedge j.\texttt{r} = k \wedge j.\texttt{w} = k \wedge j.\texttt{v} = v \wedge \texttt{abs\_valsRR} = V \cup \{v\} \wedge \\
\texttt{count\_w}[k] = c+1 \wedge \texttt{proph\_w}[k] = \texttt{false} \wedge \texttt{abs\_res\_w}[k] = \texttt{false},
\end{array}
$$

$(\texttt{AckWrite2})_j$

$$
\begin{array}{l}
reqWR(i,j,k,v,m) \wedge i = ((k-1) \bmod n) + 1 \wedge v \neq \texttt{undef} \wedge k \geq j.\texttt{r} = r \wedge \\
\texttt{abs\_res\_w}[k] = ar \wedge \texttt{proph\_w}[k] = \texttt{undef} \wedge \texttt{count\_w}[k] = c \\
\rightsquigarrow \\
ackWR(j,i,k,v,m) \wedge j.\texttt{r} = k \wedge j.\texttt{w} = k \wedge j.\texttt{v} = v \wedge \\
\texttt{count\_w}[k] = c+1 \wedge \texttt{proph\_w}[k] = \texttt{undef} \wedge \texttt{abs\_res\_w}[k] = ar,
\end{array}
$$

and $(\texttt{AckWrite3})_j$

$$
\begin{array}{l}
reqWR(i,j,k,v,m) \wedge i = ((k-1) \bmod n) + 1 \wedge v \neq \texttt{undef} \wedge k \geq j.\texttt{r} = r \wedge \\
\texttt{abs\_res\_w}[k] = ar \neq \texttt{undef} \wedge \texttt{proph\_w}[k] = p \wedge \texttt{count\_w}[k] = c \\
\rightsquigarrow \\
ackWR(j,i,k,v,m) \wedge j.\texttt{r} = k \wedge j.\texttt{w} = k \wedge j.\texttt{v} = v \wedge \\
\texttt{count\_w}[k] = c+1 \wedge \texttt{proph\_w}[k] = p \wedge \texttt{abs\_res\_w}[k] = ar.
\end{array}
$$

The guarantee relation for the code of acceptor j is the one induced by the union of these actions as follows:

$$\begin{aligned}(\texttt{Acceptor})_j \equiv \bigcup_k (\,&(\texttt{Receive})_{(j,((k-1) \bmod n)+1,[\texttt{RE},k])} \cup \\ &(\texttt{Send})_{(((k-1) \bmod n)+1,j,[\texttt{nackRE},k])}) \cup \\ &(\texttt{Send})_{(((k-1) \bmod n)+1,j,[\texttt{nackWR},k])}) \cup \\ &\bigcup_{k,v\neq\texttt{undef}}((\texttt{Receive})_{(j,((k-1) \bmod n)+1,[\texttt{WR},k,v])}) \cup \\ &(\texttt{ReadSucceeds2})_j \cup (\texttt{AckRead1})_j \cup (\texttt{AckRead2})_j \cup \\ &(\texttt{AckRead3})_j \cup \\ &(\texttt{WriteSucceeds})_j \cup (\texttt{AckWrite1})_j \cup (\texttt{WriteFails2})_j \cup \\ &(\texttt{AckWrite2})_j \cup (\texttt{AckWrite3})_j.\end{aligned}$$

The rely relation for both `read(k)` and `write(k, vW)` is

$$\bigcup_j (\texttt{Acceptor})_j \cup \bigcup_{k\neq\texttt{k}} (\texttt{Reader})_k \cup \bigcup_{k\neq\texttt{k},v\neq\texttt{undef}} (\texttt{Writer})_{(k,v)},$$

and rely relation for the code of acceptor `j` is

$$\bigcup_{j\neq\texttt{j}} (\texttt{Acceptor})_j \cup \bigcup_k (\texttt{Reader})_k \cup \bigcup_{k,v\neq\texttt{undef}} (\texttt{Writer})_{(k,v)}.$$

The proof outline below helps to show that if $AbsRR \wedge InvRR$ holds at the beginning of a method invocation, for both `read(k)` and `write(k, vW)`, then it also holds and the end of the method invocation after the abstract operation has been performed at the linearisation point, and that the abstract result (`abs_res_r[k]` and `abs_res_w[k]` respectively) coincide with the result of the concrete method. It also helps to show that each method ensures the corresponding guarantee relation, this is, the states between any atomic operation are in the guarantee relation.

```
1  val abs_vRR := undef;
2  int abs_round := 0;
3  set of val abs_valsRR := {undef};
4  val abs_res_r[1..∞] := undef;
5  val abs_res_w[1..∞] := undef;
6  int count_r[1..∞] := 0;
7  int count_w[1..∞] := 0;
8  (bool × val) proph_r[1..∞] := undef;
9  bool proph_w[i..∞] := undef;
10
11 read(int k) {
12   int j; val v; int kW; val maxV; int maxKW; set of int Q; msg m;
13   assume(pid() = ((k - 1) mod n) + 1);
14   {pid() = ((k − 1) mod n) + 1 ∧ AbsRR ∧ InvRR}
15   ⟨ if (operation reaches PL: RE_SUCC and define v = maxV at that time) {
16       proph_r[k] := (true, v); }
17     else { if (operation reaches PL: RE_FAIL) {
18       proph_r[k] := (false, _); } } ⟩
19   {pid() = ((k − 1) mod n) + 1 ∧ AbsRR ∧ InvRR}
20   for (j := 1, j <= n, j++) { send(j, [RE, k]); }
21   {pid() = ((k − 1) mod n) + 1 ∧ AbsRR ∧ InvRR}
```

```
22  maxKW := 0; maxV := undef; Q := {};
23  {maxKW = 0 ∧ maxV = undef ∧
    count_r[k] ≥ |Q| ∧ pid() = ((k − 1) mod n) + 1 ∧ AbsRR ∧ InvRR}
24  do {
25    {maxKW = max({k′ | ∃(j, v, m). j ∈ Q ∧ ackRE(j, i, k, v, k′, m)} ∪ {0}) ∧
      (maxKW = 0 ∨ (∃(j, m). j ∈ Q ∧ ackRE(j, i, k, maxV, maxKW, m)) ∧
      count_r[k] ≥ |Q| ∧ i = pid() = ((k − 1) mod n) + 1 ∧ AbsRR ∧ InvRR}
26    (j, m) := receive();
27    {sent(j, i, m, m) ∧ received(i, j, m, m) ∧
      maxKW = max({k′ | ∃(j, v, m). j ∈ Q ∧ ackRE(j, i, k, v, k′, m)} ∪ {0}) ∧
      (maxKW = 0 ∨ (∃(j, m). j ∈ Q ∧ ackRE(j, i, k, maxV, maxKW, m)) ∧
      count_r[k] > |Q| ∧ i = pid() = ((k − 1) mod n) + 1 ∧ AbsRR ∧ InvRR}
28    switch (m) {
29      case [ackRE, @k, v, kW]:
30        {ackRE(j, i, k, v, kW, m) ∧
          maxKW = max({k′ | ∃(j, v, m). j ∈ Q ∧ ackRE(j, i, k, v, k′, m)} ∪ {0}) ∧
          (maxKW = 0 ∨ (∃(j, m). j ∈ Q ∧ ackRE(j, i, k, maxV, maxKW, m)) ∧
          count_r[k] > |Q| ∧ i = pid() = ((k − 1) mod n) + 1 ∧ AbsRR ∧ InvRR}
31        Q := Q ∪ {j};
32        {j ∈ Q ∧ ackRE(j, i, k, v, kW, m) ∧
          maxKW = max({k′ | ∃(j, v, m). j ∈ Q ∧ ackRE(j, i, k, v, k′, m)} ∪ {0}) ∧
          (maxKW = 0 ∨ (∃(j, m). j ∈ Q ∧ ackRE(j, i, k, maxV, maxKW, m)) ∧
          count_r[k] ≥ |Q| ∧ i = pid() = ((k − 1) mod n) + 1 ∧ AbsRR ∧ InvRR}
33        if (kW >= maxKW) {
34          {kW ≥ maxKW ∧ j ∈ Q ∧ ackRE(j, i, k, v, kW, m) ∧
            maxKW = max({k′ | ∃(j, v, m). j ∈ Q ∧ ackRE(j, i, k, v, k′, m)} ∪ {0}) ∧
            (maxKW = 0 ∨ (∃(j, m). j ∈ Q ∧ ackRE(j, i, k, maxV, maxKW, m)) ∧
            count_r[k] ≥ |Q| ∧ i = pid() = ((k − 1) mod n) + 1 ∧ AbsRR ∧ InvRR}
35          maxKW := kW; maxV := v;
36          {maxKW = kW ∧ maxV = v ∧ j ∈ Q ∧ ackRE(j, i, k, v, kW, m) ∧
            maxKW = max({k′ | ∃(j, v, m). j ∈ Q ∧ ackRE(j, i, k, v, k′, m)} ∪ {0}) ∧
            (maxKW = 0 ∨ (∃(j, m). j ∈ Q ∧ ackRE(j, i, k, maxV, maxKW, m)) ∧
            count_r[k] ≥ |Q| ∧ i = pid() = ((k − 1) mod n) + 1 ∧ AbsRR ∧ InvRR}
37        }
38        {j ∈ Q ∧ ackRE(j, i, k, v, kW, m) ∧
          maxKW = max({k′ | ∃(j, v, m). j ∈ Q ∧ ackRE(j, i, k, v, k′, m)} ∪ {0}) ∧
          (maxKW = 0 ∨ (∃(j, m). j ∈ Q ∧ ackRE(j, i, k, maxV, maxKW, m)) ∧
          count_r[k] ≥ |Q| ∧ i = pid() = ((k − 1) mod n) + 1 ∧ AbsRR ∧ InvRR}
39      case [nackRE, @k]:
40        {proph_r[k] = (false, _) ∧ abs_res_r[k] = undef ∧
          maxKW = max({k′ | ∃(j, v, m). j ∈ Q ∧ ackRE(j, i, k, v, k′, m)} ∪ {0}) ∧
          (maxKW = 0 ∨ (∃(j, m). j ∈ Q ∧ ackRE(j, i, k, maxV, maxKW, m)) ∧
          count_r[k] ≥ |Q| ∧ i = pid() = ((k − 1) mod n) + 1 ∧ AbsRR ∧ InvRR}
41        ⟨ linRE(k, undef, false); proph_r[k] := undef;
42          return (false, _); ⟩ // PL: RE_FAIL
43    }
```

```
44    {j ∈ Q ∧ ackRE(j, i, k, v, kW, m) ∧
      maxKW = max({k' | ∃(j, v, m). j ∈ Q ∧ ackRE(j, i, k, v, k', m)} ∪ {0}) ∧
      (maxKW = 0 ∨ (∃(j, m). j ∈ Q ∧ ackRE(j, i, k, maxV, maxKW, m)) ∧
      count_r[k] ≥ |Q| ∧ i = pid() = ((k − 1) mod n) + 1 ∧ AbsRR ∧ InvRR}
45    if (|Q| = ⌈(n+1)/2⌉) {
46      {proph_r[k] = (true, maxV) ∧ j ∈ Q ∧ ackRE(j, i, k, v, kW, m) ∧
        maxKW = max({k' | ∃(j, v, m). j ∈ Q ∧ ackRE(j, i, k, v, k', m)} ∪ {0}) ∧
        (maxKW = 0 ∨ (∃(j, m). j ∈ Q ∧ ackRE(j, i, k, maxV, maxKW, m)) ∧
        count_r[k] ≥ |Q| ∧ i = pid() = ((k − 1) mod n) + 1 ∧ AbsRR ∧ InvRR}
47      return (true, maxV); // PL: RE_SUCC
48    }
49    {maxKW = max({k' | ∃(j, v, m). j ∈ Q ∧ ackRE(j, i, k, v, k', m)} ∪ {0}) ∧
      (maxKW = 0 ∨ (∃(j, m). j ∈ Q ∧ ackRE(j, i, k, maxV, maxKW, m)) ∧
      count_r[k] ≥ |Q| ∧ i = pid() = ((k − 1) mod n) + 1 ∧ AbsRR ∧ InvRR}
50  } while (true); }
51
52  write(int k, val vW) {
53    int j; set of int Q; msg m;
54    assume(!(vW = undef));
55    assume(pid() = ((k - 1) mod n) + 1);
56    {pid() = ((k − 1) mod n) + 1 ∧ vW ≠ undef ∧ AbsRR ∧ InvRR}
57    ⟨ if (operation reaches PL: WR_SUCC) { proph_w[k] := true; }
58      else { if (operation reaches PL: WR_FAIL) {
59               proph_w[k] := false; } } ⟩
60    {pid() = ((k − 1) mod n) + 1 ∧ vW ≠ undef ∧ AbsRR ∧ InvRR}
61    for (j := 1, j <= n, j++) { send(j, [WR, k, vW]); }
62    {pid() = ((k − 1) mod n) + 1 ∧ vW ≠ undef ∧ AbsRR ∧ InvRR}
63    Q := {};
64    {count_w[k] ≥ |Q| ∧ pid() = ((k − 1) mod n) + 1 ∧ vW ≠ undef ∧ AbsRR ∧ InvRR}
65    do {
66      {count_w[k] ≥ |Q| ∧ pid() = ((k − 1) mod n) + 1 ∧ vW ≠ undef ∧ AbsRR ∧ InvRR}
67      (j, m) := receive();
68      {sent(j, i, m) ∧ received(i, j, m) ∧ count_w[k] ≥ |Q| ∧
        i = pid() = ((k − 1) mod n) + 1 ∧ vW ≠ undef ∧ AbsRR ∧ InvRR}
69      switch (m) {
70        case [ackWR, @k]:
71          {ackWR(j, i, k, vW) ∧ count_w[k] > |Q| ∧
          i = pid() = ((k − 1) mod n) + 1 ∧ vW ≠ undef ∧ AbsRR ∧ InvRR}
72          Q := Q ∪ {j};
73          {ackWR(j, i, k, vW) ∧ count_w[k] ≥ |Q| ∧
          i = pid() = ((k − 1) mod n) + 1 ∧ vW ≠ undef ∧ AbsRR ∧ InvRR}
74        case [nackWR, @k]:
75          {proph_r[k] = fail ∧ count_w[k] ≥ |Q| ∧
          pid() = ((k − 1) mod n) + 1 ∧ vW ≠ undef ∧ AbsRR ∧ InvRR}
76          ⟨ if (count_w[k] = 0) {
77              linWR(k, vW, false); proph_w[k] := undef; }
78            return false; ⟩ // PL: WR_FAIL
79      }
80      {count_w[k] ≥ |Q| ∧ pid() = ((k − 1) mod n) + 1 ∧ vW ≠ undef ∧ AbsRR ∧ InvRR}
```

```
81      if (|Q| = ⌈(n+1)/2⌉) {
82        {proph_r[k] = true ∧ count_w[k] ≥ ⌈(n+1)/2⌉ ∧
          pid() = ((k − 1) mod n) + 1 ∧ vW ≠ undef ∧ AbsRR ∧ InvRR}
83        return true; // PL: WR_SUCC
84      }
85      {count_w[k] ≥ |Q| ∧ pid() = ((k − 1) mod n) + 1 ∧ vW ≠ undef ∧ AbsRR ∧ InvRR}
86    } while (true); }
```

The proof outline below helps to show that the code of each process acceptor meets the invariant $AbsRR \wedge InvRR$ and also ensures the guarantee relation, this is, the states between any atomic operation are in the corresponding guarantee relation.

```
1   process Acceptor(int j) {
2     val v := undef; int r := 0; int w := 0;
3     start() {
4       int i; msg m; int k;
5       do {
6         {pid() = j ∧ AbsRR ∧ InvRR}
7         (i, m) := receive();
8         {sent(i, j, m, m) ∧ received(j, i, m, m) ∧
          i = ((k − 1) mod n) + 1 ∧ pid() = j ∧ AbsRR ∧ InvRR}
9         switch (m) {
10          case [RE, k]:
11            {reqRE(i, j, k, m) ∧ i = ((k − 1) mod n) + 1 ∧ pid() = j ∧ AbsRR ∧ InvRR}
12            if (k < r) {
13              {k < r ∧ reqRE(i, j, k, m) ∧
                i = ((k − 1) mod n) + 1 ∧ pid() = j ∧ AbsRR ∧ InvRR}
14              send(i, [nackRE, k]);
15              {sent(j, i, ('nackRE', k, _, _), m) ∧ k < r ∧ reqRE(i, j, k, m) ∧
                i = ((k − 1) mod n) + 1 ∧ pid() = j ∧ AbsRR ∧ InvRR}
16            }
17            else {
18              {k ≥ r ∧ reqRE(i, j, k, m) ∧
                i = ((k − 1) mod n) + 1 ∧ pid() = j ∧ AbsRR ∧ InvRR}
19              ⟨ r := k;
20                if (abs_res_r[k] = undef) {
21                  if (proph_r[k] = (true, v)) {
22                    if (count_r[k] = ⌈(n+1)/2⌉ - 1) {
23                      linRE(k, v, true); } } }
24                count_r[k]++; send(i, [ackRE, k, v, w]);
25                {((count_r[k] = ⌈(n + 1)/2⌉ ∧ proph_r[k] = (true, v) ∧
                    abs_roundRR ≤ k ∧ abs_vRR = v ∧ abs_res_r[k] = (true, v))
                  ∨ ((count_r[k] ≠ ⌈(n + 1)/2⌉ ∨ proph_r[k] ≠ (true, _)) ∧
                      abs_res_r[k] = undef))
                  ∨ abs_res_r[k] ≠ undef) ∧
                count_r[k] > 0 ∧ r = k ∧ ackRE(j, i, k, v, w, m) ∧
                i = ((k − 1) mod n) + 1 ∧ pid() = j ∧ AbsRR ∧ InvRR}
26              ⟩
```

```
27          }
28          {pid() = j ∧ AbsRR ∧ InvRR}
29        case [WR, k, vW]:
30          {reqWR(i, j, k, vW, m) ∧ vW ≠ undef ∧
            i = ((k − 1) mod n) + 1 ∧ pid() = j ∧ AbsRR ∧ InvRR}
31          if (k < r) {
32            {k < r ∧ reqWR(i, j, k, vW, m) ∧ vW ≠ undef ∧
              i = ((k − 1) mod n) + 1 ∧ pid() = j ∧ AbsRR ∧ InvRR}
33            send(j, i, [nackWR, k]);
34            {sent(j, i, [nackWR, k], m) ∧ k < r ∧ reqWR(i, j, k, vW, m) ∧
              vW ≠ undef ∧ i = ((k − 1) mod n) + 1 ∧ pid() = j ∧ AbsRR ∧ InvRR}
35          }
36          else {
37            {k ≥ r ∧ reqWR(i, j, k, vW, m) ∧ vW ≠ undef ∧
              i = ((k − 1) mod n) + 1 ∧ pid() = j ∧ AbsRR ∧ InvRR}
38            ⟨ r := k; w := k; v := vW;
39              if (abs_res_w[k] = undef) {
40                if (!(proph_w[k] = undef)) {
41                  if (proph_w[k]) {
42                    if (count_w[k] = ⌈(n+1)/2⌉ - 1) {
43                      linWR(k, vW, true); } }
44                  else { linWR(k, vW, false); } } }
45              count_w[k]++; send(j, i, [ackWR, k]);
46              {((count_w[k] = ⌈(n + 1)/2⌉ ∧ proph_w[k] = true ∧
                abs_roundRR ≥ k ∧ abs_vRR = vW ∧ abs_res_w[k] = true)
                ∨ (count_w[k] ≠ ⌈(n + 1)/2⌉ ∧ proph_w[k] = true ∧
                  abs_res_w[k] = undef)
                ∨ (proph_w[k] = false ∧ abs_res_w[k] = false)
                ∨ proph_w[k] = undef ∨ abs_res_w[k] ≠ undef) ∧
              count_w[k] > 0 ∧ r = w = k ∧ v = vW ∧ ackWR(j, i, k, vW, m) ∧
              vW ≠ undef ∧ i = ((k − 1) mod n) + 1 ∧ pid() = j ∧ AbsRR ∧ InvRR}
47            ⟩
48          }
49          {pid() = j ∧ AbsRR ∧ InvRR}
50      }
51      {pid() = j ∧ AbsRR ∧ InvRR}
52    } while (true); }
53  }
```

□

D Encoding SD-Paxos as an Abstract Protocol

Let Prog be the set of programs of a language that subsumes the imperative while language for our pseudo-code in Sections 3 and 4, and which adds a parallel composition operator ‖, which is commutative and associative, and a null process 0, which is the neutral element of ‖. The semantics of the parallel composition operator that we need here is very simplistic and it does not pose any issue regarding the interaction of the components within the parallel composition.

(The interaction will be implemented on top of this operational semantics by the network semantics of the abstract distributed protocols introduced in Section 5.) The only purpose of $\|$ here is to have processes that adopt both the roles of acceptor and proposer, and to allow any of these two roles to make a move. The language Prog is morally a sequential programming language.

We let $\mathsf{Nodes} = \mathbb{N}$ be the set of natural numbers, and $\mathcal{M}.\mathit{content}$ be the set of contents of the messages in the message vocabulary $\mathcal{M}$, which contains requests for read and write, and their corresponding acknowledgements and non-acknowledgements as described in Section 3.

Now we assume a small-step operational semantics à la Winskel [30] for the programs in Prog. We let the relation $\xRightarrow[\text{int}]{i}: (\mathsf{Prog} \times \Delta_{\mathsf{nod}}) \times (\mathsf{Prog} \times \Delta_{\mathsf{nod}})$ be given by the operational semantics of a program run by node i whose current program line is not any of the network operations `send` or `receive` and where Δ_{nod} is the set of local states. We fix relations $\xRightarrow[\text{snd}]{i}: (\mathsf{Prog} \times \Delta_{\mathsf{nod}}) \times (\mathsf{Prog} \times \Delta_{\mathsf{nod}} \times \mathcal{M})$ and $\xRightarrow[\text{rcv}]{i}: (\mathsf{Prog} \times \Delta_{\mathsf{nod}} \times \mathcal{M}) \times (\mathsf{Prog} \times \Delta_{\mathsf{nod}})$ to be the ones induced by the rules:

SEND

$$\frac{P = (\texttt{send}(J, C); P_1) \parallel P_2 \qquad P' = P_1 \parallel P_2 \qquad m.\mathit{to} = J \qquad m.\mathit{content} = C \qquad m.\mathit{from} = i}{\langle P, \delta\rangle \xRightarrow[\text{snd}]{i} \langle P', \delta, m\rangle}$$

RECEIVE

$$\frac{P = ((J, C) \texttt{ := receive}(); P_1) \parallel P_2 \qquad P' = P_1 \parallel P_2 \qquad m.\mathit{content} = c \qquad m.\mathit{from} = j \qquad m.\mathit{to} = i \qquad \delta' = \delta[J \mapsto j, C \mapsto c]}{\langle P, \delta, m\rangle \xRightarrow[\text{rcv}]{i} \langle P', \delta'\rangle}$$

The J and the C in rule SEND above are meta-variables for some expressions of Prog with types Nodes and $\mathcal{M}.\mathit{content}$ respectively. In rule RECEIVE, there is an assignment and the J and the C this time are meta-variables for names of some fields in the local state $\delta \in \Delta_{\mathsf{nod}}$, which we assume have types Nodes and $\mathcal{M}.\mathit{content}$ respectively (*e.g.*, in our implementation of SD-Paxos in Figure 3, J and C are respectively substituted by `j` and `[RE, k]` in line 5, and by `j` and `m` in line 8).

Next, we will fix a particular set of local states Δ_{nod} that matches with our implementation of SD-Paxos in Section 3, and we will derive an abstract distributed protocol for SD-Paxos from the relations defined in the previous paragraph. We write $\mathbb{B} = \{\mathsf{True}, \mathsf{False}\}$ for the set of Booleans and $\mathbb{V}$ for the set of values that are decided by Paxos. We distinguish two sets of local state, Δ_{acc} and Δ_{pro}, for acceptors and proposers respectively:

$$\Delta_{\mathsf{acc}} = \{\texttt{j} : \mathsf{Nodes};\ \texttt{v} : \mathbb{V};\ \texttt{r} : \mathbb{N};\ \texttt{w} : \mathbb{N};\ \texttt{i} : \mathsf{Nodes};\ \texttt{m} : \mathcal{M}.\mathit{content};\ \texttt{k} : \mathbb{N}\}$$

$$\Delta_{\mathsf{pro}} = \{\texttt{v0} : \mathbb{V};\ \texttt{kP} : \mathbb{N};\ \texttt{resP} : \mathbb{B};\ \texttt{vP} : \mathbb{V};\ \texttt{resRC} : \mathbb{B};\ \texttt{vRC} : \mathbb{V};\ \texttt{jRR} : \mathsf{Nodes};\ \texttt{QRR} : \{\mathsf{Nodes}\};\ \texttt{mRR} : \mathcal{M}.\mathit{content};\ \texttt{vRE} : \mathbb{V};\ \texttt{kW} : \mathbb{N};\ \texttt{maxV} : \mathbb{V};\ \texttt{maxKW} : \mathbb{N}\}$$

A local state for an acceptor $\delta \in \Delta_{\mathsf{acc}}$ contains a copy of the parameters and fields of process `Acceptor` and the local variables of its task `start()` in

Figure 4 of Section 3. A local state for a proposer $\delta \in \Delta_{\mathsf{pro}}$ contains a copy of the parameters and the local variables of the client code in Figures 3 and 5 of Section 3. For simplicity, we flatten the code of `proposeP` on the right of Figure 5 by inlining the codes of `proposeRC`, `read` and `write`. To avoid clashing names, we have appended one of the suffixes `P`, `RC`, `RR` or `RE` to the names of some of the variables, and we have used top-most variables instead of parameters of inlined methods when possible. For instance, fields `vP`, `vRC` and `vRE` correspond respectively to the variable `v` in each of the methods `proposeP`, `proposeRC` and `read`, field `v0` is used in place of the variable with the same name in both `proposeP` and `proposeRC`, field `kP` is used in place of the variable `k` in every method, and field `vRC` is used in place of the variable `vW` in `write`. We have also reused fields `jRR`, `QRR` and `mRR` in place of variables `j`, `Q` and `m` in both `read` and `write`, since these two methods are invoked sequentially and do not interfere with each other.

For each acceptor i with local state $\delta \in \Delta_{\mathsf{acc}}$, we tag the node by letting $\delta.\mathtt{role} = \mathit{Acceptor}$ and we let the implicit filed $\delta.\mathtt{pid}$ coincide with i and with the field $\delta.\mathtt{j}$. Customarily, all the nodes are acceptors. If, besides, i is a proposer that proposes value v, then its local state δ is in $\Delta_{\mathsf{acc}} \times \Delta_{\mathsf{pro}}$ and we tag the node $\delta.\mathtt{role} = \mathit{Proposer}$ and additionally we let $\delta.\mathtt{v0} = v$. The set of local states for the operational semantics of our implementation of SD-Paxos is

$$\Delta_{\mathsf{nod}} = (\Delta_{\mathsf{acc}} + (\Delta_{\mathsf{acc}} \times \Delta_{\mathsf{pro}}))$$

Now we can fix the start configurations for the operational semantics. For each acceptor i that is not a proposer, the initial local state is

$$\delta^i_{\mathsf{acc}}[\mathtt{j} \mapsto i, \mathtt{v} \mapsto \bot, \mathtt{r} \mapsto 0, \mathtt{w} \mapsto 0] \in \Delta_{\mathsf{acc}}$$

and the relation $\xRightarrow[\text{int}]{i}$ starts at the configuration $\langle \mathtt{start()} \parallel 0, \delta^i_{\mathsf{acc}} \rangle$, where `start()` is the code of an acceptor in Figure 4 and 0 is the null process.

For each proposer i that proposes value v, the initial local state is

$$\delta^i_{\mathsf{pro}}[\mathtt{j} \mapsto i, \mathtt{v} \mapsto \bot, \mathtt{r} \mapsto 0, \mathtt{w} \mapsto 0, \mathtt{v0} \mapsto v] \in \Delta_{\mathsf{acc}} \times \Delta_{\mathsf{pro}}$$

and the relation $\xRightarrow[\text{int}]{i}$ starts at the configuration $\langle \mathtt{start()} \parallel \mathtt{proposeP}, \delta^i_{\mathsf{pro}} \rangle$, where `proposeP` is the client code obtained by flattening the codes in Figures 3 and 5 by inlining `proposeRC`, `read` and `write`, as explained in the previous paragraphs.

Now we can define an *SD-Paxos sequence of a node* i as a sequence of configurations $\langle P_0, \delta_0 \rangle$, $\langle P_1, \delta_1 \rangle$, ... in $\mathsf{Prog} \times \Delta_{\mathsf{nod}}$ such that

- $\langle P_0, \delta_0 \rangle = \langle \mathtt{start()} \parallel 0, \delta^i_{\mathsf{acc}} \rangle$ if the node i is an acceptor that is not a proposer, or otherwise $\langle P_0, \delta_0 \rangle = \langle \mathtt{start()} \parallel \mathtt{proposeP}, \delta^i_{\mathsf{pro}} \rangle$ if node i is a proposer, and
- for every n then either $\langle P_n, \delta_n \rangle \xRightarrow[\text{int}]{i} \langle P_{n+1}, \delta_{n+1} \rangle$, or there exists a message m such that $\langle P_n, \delta_n, m \rangle \xRightarrow[\text{snd}]{i} \langle P_{n+1}, \delta_{n+1} \rangle$ or $\langle P_n, \delta_n \rangle \xRightarrow[\text{rcv}]{i} \langle P_{n+1}, \delta_{n+1}, m \rangle$.

The encoding of SD-Paxos as an abstract distributed protocol is code-aware, this is, the text of the program that a particular process is running is included in the local state. The text of the program provides information about the program flow, and saves us from using auxiliary machinery to represent control-related aspects. The local state for the abstract distributed protocol is

$$\Delta = \mathsf{Prog} \times \Delta_{\mathsf{nod}}$$

We let $\sigma : \mathsf{Nodes} \rightharpoonup \Delta$ be such that for each process identifier i in the range of 1 to n, the local state of node i is $\delta = \sigma(i)$. Notice that the implicit field $\delta.\texttt{pid}$ coincides with the origin i of the mapping $[i \mapsto \delta] \in \sigma$.

Now we can define the relations $\mathcal{S}_{\text{int}}$, $\mathcal{S}_{\text{snd}}$ and $\mathcal{S}_{\text{rcv}}$ in terms of an SD-Paxos sequence of a node.

A pair $\langle \delta, \delta' \rangle$ is in $\mathcal{S}_{\text{int}}$ iff either $\delta = \delta'$, or otherwise there exist i such that δ and δ' are consecutive steps in some SD-Paxos sequence of node i and $\delta \xRightarrow[\text{int}]{i} \delta'$.

A triple $\langle \delta, \delta', \{m\} \rangle$ is in $\mathcal{S}_{\text{snd}}$ iff there exist i such that δ and δ' are consecutive steps in some SD-Paxos sequence of node i and $\langle \delta, m \rangle \xRightarrow[\text{snd}]{i} \delta'$.

A triple $\langle \delta, m, \delta' \rangle$ is in $\mathcal{S}_{\text{rcv}}$ iff there exist i such that δ and δ' are consecutive steps in some SD-Paxos sequence of node i and $\delta \xRightarrow[\text{rcv}]{i} \langle \delta', m \rangle$.

E Proofs of Lemmas and Theorems in Section 5

Lemma 1 (OTA refinement). *$\mathcal{B}_{\xRightarrow{p} \cup \xRightarrow[ota]{p,\mathcal{P}}} \subseteq \mathcal{B}_p$, where p is an instance of the module Paxos, as defined in Section 3 and in Example 1.*

Proof. By definition of the protocol in Example 1 a read request does not change a proposer's local state and does not have a non-trivial precondition. □

Lemma 2 (Slot-replicating simulation). *For all $I, i \in I$, $\mathcal{B}_\times|_i = \mathcal{B}_p$.*

Proof. Both ways: by induction on the length of the target behaviour, taking into the account that we can always add "stuttering" states. □

Lemma 3. *If T from* WStepReceiveT *is OTA-compliant with predicate $\mathcal{P}$, such that $\mathcal{B}_{\xRightarrow{p} \cup \xRightarrow[ota]{p,\mathcal{P}}} \subseteq \mathcal{B}_{\xRightarrow{p}}$ and p is $\mathcal{P}$-monotone, then $\mathcal{B}_{\xRightarrow{\triangledown}} \subseteq \mathcal{B}_{\xRightarrow{\times}}$.*

Proof. The proof is by induction on the length of the trace. The rule WStepReceiveT can replicate a message $m \in T$ for multiple slots $j \in I$. To match adding a replica of m for each j with the corresponding per-slot executions of the simple semantics, we use Definitions 3 and 4 to show that for any $j \in I$ we can relate its replica of m to the current local state (due to p's $\mathcal{P}$-monotonicity). This allows us to emulate this step by a per-slot execution of the OTASend. The result then follows from the refinement assumption. □

Theorem 4. *The implementation of Multi-Paxos that uses a register provider and bunching network semantics refines the specification in Figure 17.*

Proof. The theorem is a straightforward consequence of the linearisation result stated by Theorem 1 and of the the results about the transformations of the network semantics stated by Lemmas 1 to 5. □